Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
2016

Ricardo Filipe Gonçalves Ribeiro

TASKA: Sistema modular e extensível para fluxos de trabalho repetíveis

TASKA: A modular and easily extendable system for repeatable workflows

Universidade de Aveiro
2016
Departamento de Electrónica, Telecomunicações e Informática

Ricardo Filipe Gonçalves Ribeiro

TASKA: Sistema modular e extensível para fluxos de trabalho repetíveis

TASKA: A modular and easily extendable system for repeatable workflows

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor José Luís Guimarães Oliveira, Professor associado do Departamento de Electrónica. Telecomunicações e Informática da Universidade de Aveiro

o júri

presidente

Prof. Doutor José Manuel Matos Moreira
professor auxiliar da Universidade de Aveiro

Prof. Doutor Rui Pedro Sanches de Castro Lopes
professor coordenador do Instituto Politécnico de Bragança

Prof. Doutor José Luís Guimarães Oliveira
professor associado da Universidade de Aveiro

agradecimentos

Antes de mais, gostaria de agradecer à minha família, especialmente à minha esposa, pela paciência com que aturou os meus devaneios durante o desenvolvimento deste trabalho. Gostaria também de agradecer ao grupo de bioinformática, nomeadamente ao meu orientador, o professor José Luís Oliveira e ao meu colega Luís Bastião, sem os quais este trabalho não teria sido possível.

acknowledgements

First of all, I would like to thank my family, specially my wife, for the patience to put up with my ramblings during the development of this work. I would also like to thank the bioinformatics group, my thesis supervisor José Luís Oliveira and my colleague Luís Bastião. Without them this work would not have been possible.

palavras-chave

Fluxo; Trabalho; Tarefa; Web; SaaS; Webservice; Python; Javascript; reactJS; Django; EMIF

resumo

O tempo é um dos recursos mais importantes. As tarefas diárias consomem tempo, e grande parte do nosso esforço como seres humanos é dedicado ao melhoramento da eficiência de processos repetitivos do dia-a-dia. Embora grande parte do esforço já esteja feito, existe ainda uma falha de soluções a nível de gestão de fluxos de trabalho, que englobe trabalho colaborativo preencha requisitos complexos e apresente uma interface moderna e atual baseada em componentes web.

Este trabalho descreve uma solução web para gestão de fluxos de trabalho, denominada “Taska”, que pode ser usada em várias áreas de otimização de processos. O trabalho apresentada nesta tese pretende ser um sistema híbrido que cobre não só gestão de fluxos de trabalho, mas também permite gerir, guardar e partilhar documentação associada com processos e tarefas. O sistema foi desenhado com uma estrutura extensível e modular e pode ser usado como uma plataforma “Software como Serviço”. Por um lado, todo o sistema pode ser utilizado por aplicações externas para gerir fluxos de trabalho complexos e manter em registo máquinas de estado, processos e tarefas. Por outro lado, também apresentamos uma interface de utilizador web moderna, com grande usabilidade e que permite aos utilizadores criarem fluxos de trabalho interactivamente e com capacidades colaborativas entre vários parceiros/colaboradores.

O “Taska” foi desenvolvido para preencher uma falha no projeto Europeu EMIF, e tem sido usado como estudo piloto por investigadores translacionais, e proprietários de base de dados para facilitar o seu trabalho em estudos de saúde em base de dados dispersas à volta do mundo.

keywords

Workflow; Task; Web; SaaS; Webservice; Python; Javascript; reactJS; Django; EMIF

abstract

Time is one of the most important resources in the world. Daily tasks consumes time, and much of the human beings efforts are dedicated to improve the efficiency of repetitive processes of our daily life. While many of the efforts has been developed already, there are still a gap in the workflow management processes, mainly due to collaborative work, high complexity requirements and missing a modern and actual user experience through the new web components. This work describes a web solution for workflow management, named “Taska” that can be used in many areas of process optimization. The work presented in this thesis intends to be a hybrid system that covers not only the workflow management processes, but also keep to manage, store and share the documentation associated with processes and tasks. The system was designed with a modular and extensible structure and can be used as a Software-as-a-Service platform. In one hand, all the system can be used by external applications to management complex workflows and keep recorded their state machines, processes and tasks, with capability to execute the state machines and compute the results. One another hand, we also present a modern web front-end application that is user-friendly and allow users to create complex workflows graphically and with collaborative features to perform the workflow with different partners/collaborators.

Taska was developed to fulfil the gaps in a European project EMIF, and has been used in a pilot study by translational researchers, and database owners to find facilitate their work in custom health studies in disperse databases around the world.

Contents

List of Figures

List of Tables

Acronyms

Term	Description
Ajax	Asynchronous Javascript and XML
AGPL	Affero General Public License
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
ASP	Active Server Pages
BPM	Business Process Management
BPMN	Business Process Model and Notation
BSD	Berkeley Software Distribution
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
CSV	Comma Separated Values
DBMS	Database Management System
DETI	Departamento de Electrónica, Telecomunicações e Informática
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
EJB	Enterprise Java Beans
EMIF	European Medical Information Framework
EPL	Eclipse Public License
EPUB	Electronic Publishing
ERD	Entity Relationship Diagram
ES	ECMAScript
GB	Gigabyte
GPL	Gnu Public License
GUI	Graphical user interface
HDD	Hard Disk Drive
HPC	High-Performance Computing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IEETA	Instituto de Engenharia Electrónica e Telemática de Aveiro
IO	Input/Output
IP	Internet Protocol
Java EE	Java Platform, Enterprise Edition
Java SE	Java Platform, Standard Edition
JAX-RS	Java API for RESTful Web Services
JMS	Java Message Service
JPA	Java Persistence API
JSF	JavaServer Faces
JSON	Javascript Object Notation

JVM	Java virtual machine
LDAP	Lightweight Directory Access Protocol
LGPL	GNU Lesser General Public License
MTI	Multi-Table Inheritance
MVC	Model-View-Controller
MVW	Model-View-Whatever
NPM	Node Package Manager
ORM	Object-relational mapping
PDF	Portable Document Format
PHP	PHP Hypertext Processor
PIP	Pip Installs Packages
RAM	Random-Access Memory
RDBMS	Relational DataBase Management System
REST	Representational State Transfer
SaaS	Software as a Service
SCM	Software Configuration Management
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SVN	Apache Subversion
TCP	Transmission Control Protocol
WAR	Web application ARchive
WSDL	Web Services Description Language
XLSX	Microsoft Excel Open XML Spreadsheet
XML	Extensible Markup Language
YAWL	Yet Another Workflow Language

1 Introduction

One of the most valuable resources in the world is time. Everything we do takes a determined amount of time, which has an associated cost. Therefore, citizens, companies, governmental institutions, have been seeking for better ways of reducing and optimizing the time spend in many processes, especially in those that we must repeat every day in our work and in our life.

Computers were one of the greatest tools invented for process optimization. They greatly reduce the time spend in many processes, mathematical tasks that could take years, can be now executed in a couple seconds. This revolution has been having even a bigger impact since the dawn of the Internet and all the rapid development it brought to software.

Ever since, software development has been moving toward making the maximum possible optimizations of repetitive tasks, and some of the tools that came from this goal are workflow and task management solutions.

Although it may not seem at the first look, workflow and task management system are usually very different according to the nature of the problem they try to solve. While the former usually focus on the relation between each task and the execution pipeline, dismissing the user itself, the second focus almost exclusively on the task and their users disregarding, or not emphasizing at least, the relation between the tasks and the inputs and outputs it generates and how they relate to each other.

Even though there are available some popular workflow solutions the biggest problem is they are usually targeted to very context specific, or simply not as user-friendly as one may expect.

Internet adoption almost reaches 96% on the European Union [1] and this scenario has shaped the way many software solutions are being developed. Following this trend, Web-based task manager such as Bitrix [2] and Redmine [3] have been thriving, and seem to point the way. However, they still do not nearly fulfil the requirements a workflow user needs.

One could argue there is an increasing need for a system that connects the best of both worlds (workflow and task management) and provides a smart platform that allows collaboration between different users through a friendly interface, while keeping a strong focus on the relation between the tasks that users perform and the results it produces.

It is in the light of this problem that the idea for this dissertation began to take shape. Additionally, the requirements for this work were partially defined in the context of the project “European Medical Information Framework” (EMIF), a project in which the bioinformatics group of the University of Aveiro is partner.

1.1 Goals

The current lacks of all-encompassing workflow systems, favours the development of a generic solution that brings the best of both worlds, allowing establishing a well-designed and modular first-party system on the market.

Using as a pilot the EMIF project needs for a user-friendly repeatable process management system, this dissertation aims developing a platform to solve some of the major drawbacks of existing workflow and task managers. The main goals to achieve were: 1) a web-based system that allows multiple users to collaborate on the realization of repeatable tasks and workflows in an easy and fast fashion; 2) a process manager that supports inputs, outputs and dependencies on his tasks; 3) a system that maintains a documentation log of each step of the process to review later.

1.2 Outline of thesis

This thesis is composed by six other chapters, which are briefly described below.

In chapter 2, we provide a detailed analysis of several types of process-related management systems, from commercial to open source ones, covering full-fledged applications and workflow engines, but also task managers and some very context-specific solutions for process management.

The following chapter 3 details all the platform requirements, ranging from system-wide requirements, to all the specific requirements of the pilot use-case project for the platform development.

In chapter 4, we present a brief overview of available software architectures patterns used for the implementation of software systems. As well as a quick overview of available frameworks.

In chapter 5, we present the developed solution, Taska - an extensible workflow management system. We do detailed analysis on the implementation of all components that make both the backend SaaS system as well as the frontend client, not forgetting the deploy capabilities of the system. We also board technologies and third party software used, as well as the web services and functionalities the default interfaces makes available, how they were implemented and why.

On chapter 6, we present the main results of this work, the current usage of the system, as well as user feedback from real users using the system for their business processes.

In chapter 7, we evaluate the work done, and point out some directions for further development on next versions of the software.

2 Workflow/Document Management Systems

Society is always looking for ways to make production and business processes more agile and easier to organize. The easier we get results, less time we need to spend on specific tasks and the more we can enjoy life.

A business process is related to the act of making a product (be it an actual product, or a result of some kind of work) [4], and it generally consists of a set of activities performed in coordination and in an organized way [5]. Business processes are a crucial part of any efficient organization, and as such, most environments naturally develop some kind of systemized flow of work to manage those processes. Below, on Figure 1, we can see an example of a business process.

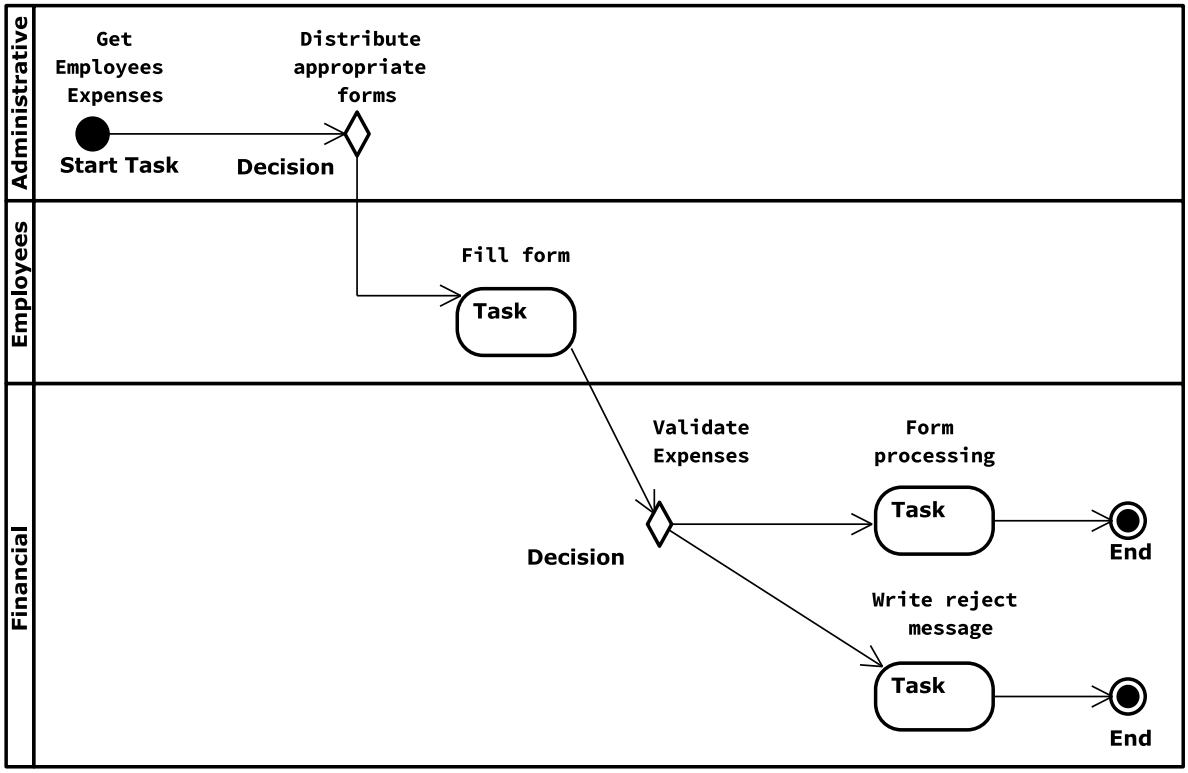

Figure 1 - Example of a business process

In a more broader definition, Business Process Management (BPM) refers to the act of design, administrate, configure, enact and analyse a business process [5]. BPM is a key tool in current companies and organizations, and several software tools can use used to accomplish it. To the generic software systems that coordinates the business processes enactment we call business process management systems [5].

Information processes are business processes that are partially automated (having assistance from a human for some part of the process) or fully automated (completely executed by a program) [6]. Workflow and Document/Task management systems are a particular kind of computer based BPM system that usually allow information processes and are the focus of this paper.

In this section, we analyse the concept and features of existing solutions, workflow management engines and workflow management languages currently available in the open source community and the market. A workflow is an orchestration of repeatable business processes that process information in a systematic fashion [7].

Workflow management platforms allow us to reengineer business and information processes, managing the business processes using different workflows for different processes, and facilitating the flow of information between all intervenient, notifying each one whenever their input is needed [6].

2.1 Full-fledged Solutions

There are a large number of workflow full-fledged systems. These platforms are ready-to-use solutions that fulfil part of the requisites of the requirements of this system.

Most part of the solutions is desktop based, and the others are web and cloud based (in the SaaS model). Usually, this kind of solutions do not allow their integration with other platforms, being mainly interface-centric. Since most of them are commercial solutions, it is not possible to extend their interface to interact or integrate with external applications.

In the sequel, follows the analysis of several workflow management solutions.

Bitrix24

Bitrix24 is a PHP Hypertext Processor (PHP) intranet solution for small and medium-sized businesses designed to be secure and ready-to-use. It provides complex features such as communication, collaboration, social networking, business process and workflow management without any hassle [2].

The platform has a couple of solutions, in different business models, such as paid per month or all-at-once, that range from self-hosted solutions to cloud-based ones. The system does not only offer task management (with subtasks, multiple participants), but also a social component with a activity stream, a workgroup chat room, a shared calendar, an intranet mail service between participants and shared documents.

One of the elements lacking in this solution is the task dependency, effectively impairing the solution for our purposes, even though it supplies many extra features. However, it does not support the most essential one required for the project: allowing a clear flow of actions that must succeed in sequence by different intervenient. Another drawback of the solution is the inexistence of an Application Program Interface (API) to interact with the system.

Change Management Control

Change Management Control is a web-based highly flexible, elegant system that allows the management of customized workflows. The system also has helpdesk functionality and can be paid monthly on cloud-based version, or once for the self-hosted version [8].

The platform fulfils most of the requirements that we desire for our proposal. It allows complex multidirectional workflow specification, with several intervenient, all over a self-hosted server or cloud-based interface. It also allows the specification of custom types of tasks, such as custom form fillings (one of our requirements).

Nevertheless, the software does not provide an API for integration with other platforms, and has incredibly high requirements to run: 100 Gigabyte (GB) of Hard Disk Drive (HDD) and 2 GB of RAM.

Taverna

Taverna is a scientific oriented workflow management system that makes available a suite of open source tools to facilitate computer simulation of repeatable scientific experiments [9].

This platform allows running over a self-hosted server and several desktop-based clients. This is a disadvantage compared to the cloud-based solutions, which also provide a web interface. On the other hand, the system is SOA, which makes several web interfaces available, such as Representational State Transfer (REST), Web Services Description Language (WSDL), BeanShell, etc. Thus, it allows us to do software integration.

The scientific community is this software's main goal. The system allows background task queuing, instead of multidirectional flows with several intervenient. The learning curve seems steeper for new users, than other platforms, not having a quick guide, or an easy deployment. We would classify it as a highly scientific and specialized platform, which is not suitable for all usages, but works well for repeatable scientific experimentation.

Stardust

Stardust is a business-process management platform based on Eclipse and licensed under the Eclipse Public License (EPL). It must be installed through the "new software" menu on Eclipse [10].

Two components mainly compose the platform. The first is a process modelling plugin on Eclipse where we model Business Process Model and Notation (BPMN) complainant workflows. The second component is an Enterprise Java Beans (EJB), JavaServer Faces (JSF) based runtime for process deployment that goes by the name of Stardust Portal and runs over any Java Platform, Enterprise Edition (Java EE) application server such as Glassfish and Tomcat. SunGard is developing this second one, which has a commercial solution named IPP (Infinity Process Portal) and donates part of the code to the Eclipse community.

There is support for several participants in a workflow, external web service integration and execution of other applications in the course of the workflow. It is also possible to simulate workflow runs and obtain reports pertaining the process execution with charts and relevant information about the process.

The project documentation is very extensive, including teaching videos and extensive reference manuals.

A big turn-off from this solution is the necessity of Eclipse to execute most of the modelling and simulation features, which are desktop-based. Although not fully implemented yet, it seems to exist an interest in developing a browser-based BPMN modeller for the platform. Our requirements pretend to integrate this on a web-based solution, which would make this a bad fit.

2.2 Workflow Engines

Conceptually, workflow engines only manage the automated aspects of a workflow process for each item determining which activities are executed next when pre-requisites are achieved attributing this tasks [11].

Workflow Management Engines do not offer ready-to-use solutions, but only the base blocks to build a workflow management system. Although this brings the obvious disadvantage of having to setup the system up, it also brings numerous vantages, mainly due to the flexibility to integrate some software modules.

For starters, these solutions are usually free (some of them open source) and most offer an SOA to integrate in other platforms, such as REST or Simple Object Access Protocol (SOAP) API'S. This simplifies the integration, making them the perfect candidate for our project.

In this sequel, we will analysis the existent workflow engines modules.

Activiti

Activiti is a rock-solid and lightweight workflow management platform strongly focused on catering the needs of business professionals, developers and system administrators [12].

The system is free and based on a Java EE deployment that allows both REST-only and REST and web interface. The software has an easy deployment and a good documentation.

The platform allows complex repeatable workflows, with different kinds of tasks (including scripting and custom forms), but only one intervenient at the time, even though it allows reassignments mid process. The system allows automatic generation of statistics over concluded workflows, such as average task duration for each workflow process, or statistics about the ending state of a repeating workflow.

FireWorks

FireWorks is an open-source scientific workflow platform focused on the management and execution of scientific workflows, intended to be friendly but flexible, allowing high computation throughputs [13]. The platform is python based, with the database running on MongoDB.

It can be used through a python API, command-line or a web-service, but only works for workflow monitoring. As with Taverna, the focus seems to be the repeatable nature of the workflows on the background, having short focus on multi-step and multi-user interactions to accomplish a task.

Most of the system focus on parallel work execution and on job scripting and processing, even having integration with popular task queuing platforms, such as Torque or Sun Grid Engine. This makes automatically the platform unsuitable for our requirements, mainly because of the thesis focus on the development of a multi user collaboration over the web platform.

FOXOpen

FOXOpen is an open-source toolkit tailored to the rapid development of workflow-based web systems [14]. The main target for this platform is the development of very complex workflow systems.

The platform is a bit different from the other engines analysed in the previous subsections, mainly because it is more of a framework than an engine. It does not give us a ready to work solution, but instead a base that to configure to create one. Thus, the features cannot be analysed, since we must implement them. It can work as a web service SOAP, and supports advanced concepts such as document generation, creation of user interfaces and automatic validation.

The hardware requirements are steep though, requiring 10 GB of disk and 1 GB of Random-Access Memory (RAM) to work, plus and oracle database installation. Big

companies with very specific needs that must have a custom-tailored system without taking into account the lost time around deployment and implementation are the intended audience.

GoFlow

GoFlow is python workflow engine that is provided in the form of a module component for the popular framework Django. It follows a Berkeley Software Distribution (BSD) licensed, and its purpose is to be integrated with other projects that use the Django framework, providing just flux control [15].

The engine is activity-based, allowing the specification of a flow between activities distributed to different users. Django role's platform is the basis for the permissions. In addition, each activity is a model instance of this activity.

Although this fulfils the basic needs of the project, and seems easy to integrate in a more complex solution, the main problems are not allowing background tasks by default, being incomplete and without support or updates since September 2008, which makes it less attractive in the future.

ViewFlow

ViewFlow is an Affero General Public License (AGPL) open-source, python-based, module for the popular Django framework that provides a workflow management library easy to integrate with other solutions [16]. The system also plans to roll out a commercial solution with additional features, later this year (2014).

Although still in it is early phase (the software is not yet in full stable release), the library seems to be promising allowing state management, permission checking, concurrent updates and synchronization by itself. The system is multi-user and allows background-jobs (celery-based) or user tasks (view-based). The views can use the normal Django format, allowing, for example, dynamic forms as part of a user task.

The main disadvantages of this solution are the fact it is still in its early stages and having high requirements technology-wise, needing python 3.3 and Django 1.7 to run, which are not the versions used on stable software on the web at the moment.

YAWL

Yet Another Workflow Language (YAWL) is a GNU Lesser General Public License (LGPL) and Java EE based, workflow system with their own powerful specification modelling language that supersets Extensible Markup Language (XML). The system runtime is based on a SOA for ease of use [17].

The system has an installer to make it easier to start using it, and runs over any Java EE application server, such as TomCat, or Glassfish.

The YAWL file specifies the workflows, but there is a visual editor written in Java Platform, Standard Edition (Java SE) that allows creating the workflows without knowing the language specification by heart. The specification is complex enough to allow multi-user collaboration in a unique overflow, based on the notion of roles and reallocation of tasks.

The workflows can be launched and managed by the users from the default web interface provided by the system, or can be used via web-services.

The documentation also appears to be very extensive and detailed, and there is ongoing support for the platform. It looks a mature project. It has been around since 2004 and would be a good base for the project.

jBPM

jBPM is an open-source, Java EE based, business process management suite that runs as a Java EE application and executes BPMN 2 repeatable workflows [18]. A H2 database maintains the persistency.

The system includes a wide range of functionalities. Like a desktop and web-based visual business process editor with support for complex tasks such as forms and a form modeller to create them to a process instance manager. The processes can also use user-modelled data schemas, obtain statistics over process instances already run and create customized dashboards to display relevant information more rapidly.

There is support for multi-user collaboration in tasks using groups of users or individual users, although I must say, I did not appreciate the way a configuration file is used to specify users and groups, with the password exposed, as opposed to a salted-approach on a database. It is a potential security risk.

2.3 Other Related Systems

There are other systems that may not be full generic workflow systems per-say, but can be related or fulfil part of the project requirements. We discuss some of these systems below.

2.3.1 Bioinformatics workflow engines

There are other systems mainly focused in bioinformatics, and on running command line tools, which are out of the scope of our system requirements, due to the lack of support for the documentation needed for the project. Below follows a description of some of them.

Yabi

Yabi is a python open source, grid based, high performance cloud computing execution environment for bioinformatics research, which provides encapsulation of data environments in an intuitive web interface [19].

The focus of the tool is to integrate data back ends with High-Performance Computing (HPC) environments in an easy way, removing the overhead of the tools setup, and learning how to use them from command-line. The system is targeted for users without the technical expertise on how to use the tools by command-line, but also system administrators who need to give simple access to their tools for other users.

The system is oriented to large-scale biomedical data analysis, without any kind of specification of multi-user workflows with user intervention being possible, and so, it does not fulfil the defined requirements.

Galaxy

Galaxy is a python cloud-based platform, oriented to computational biomedical research over big datasets [20]. Accessibility, reproducibility and transparency are the basic concepts that guided the system. Their goals are to be easy to use by users without technological knowledge, to allow repetition of experiments easily and to facilitate sharing of experiment results or analysis with other users.

Their approach is similar to Yabi, wrapping other tools in a single interface and allowing linear cascading of some of them. The social features support is interesting because it allows the discussion of other user results, and running the system ourselves to verify if they have the same results. It supplies a succinct interface, with a good workflow visual editor, but the tasks are very domain-specific and predefined which does not make it a suitable choice for our use-case.

2.3.2 Project Managers

Redmine

Redmine is an advanced and highly extendable, Gnu Public License (GPL), web project management tool written in the Ruby on Rails framework with multiple database support [3].

The array of features the tool supports by default is very large. The system is project oriented and supports per-project Software Configuration Management (SCM) integration with several well-known system as such as Apache Subversion (SVN), GIT and Mercurial, wiki pages, issue tracking with task grouping, discussion forums, Lightweight Directory Access Protocol (LDAP) authentication, multi-user collaboration and even a Gantt scheduler or a calendar.

The system features a large community of active developers, with over five hundred available plugins, making possible to extend the functionalities further like chart generation over project statistics, chat rooms, or Comma Separated Values (CSV) task exporting.

Besides the lack of task dependency and the inexistence of complex tasks (such as background scripted tasks and form tasks) the system fits quite well most of the requirements of the project, since part of the purpose of the project is documentation capabilities.

ProjectPier

ProjectPier is an intuitive, AGPL licensed, web project manager written in PHP with a MySQL database. The main objective of the project is easy online collaboration between project team members [21].

The tool is hierarchal and distributed in projects. Projects organize in milestones that contain simple tasks lists, which can have different intervenient. There is a very rudimentary discussion system and a calendar. There is the possibility to extend functionality through plugins that can add a wiki, a file repository, forms and a ticket system.

There is no task dependency besides the task list grouping and there is no complex task possibility. There is also no versioning system support. In general, the project is a bad fit because it does not provide much value and even among project managers, there are better solutions.

Taiga.io

Taiga.io is a Django-based web project manager designed mainly for code development [22]. All the features are minimalistic by design, only having a simple task manager (that organizes tasks by user stories), a basic issue tracker and a wiki. The tasks cannot be complex, and can only be marked as done/undone. There is no functionalities like calendars or Gantt charts. Comments on the user stories are the vehicle of communication between team members.

The tool is too simple and fits only the very basic requirements of the project. The objective of the tool is too much focused, and should only fit the software development paradigm.

2.4 Systems Comparison

From the discussed state-of-the-art, we conclude that the solutions already developed do not completely fulfil the needs for the workflow management system we intend to build for

our platform. On Table 1, we can see a side-by-side feature comparison of the boarded solutions.

On one hand, current full-fledged web platforms miss essential features such as allowing asynchronous tasks and easy integration with external tools. On other hand, existing workflow engines do not support multi-user features such as easy collaboration over the same workflows, discussion of the collected results, and workflow sharing between different users, effectively impairing collaboration efforts.

Our decision was to build a new web platform that allows easy collaboration between partners, with multi-user interactions and features such result discussion and workflow sharing.

Table 1 - Workflow systems features comparison

Platform Feature	Bitrix24	CMC	Taverna	Stardust	Activiti	Fireworks	FoxOpen	GoFlow	ViewFlow	Yawl	jBPM
Base Language	php	.NET	Java	Java	Java	Python	Java	Python	Python	Java	Java
License	Commercial	Commercial	Open Source	EPL	Apache 2.0	Open Source	BSD	Unknown	AGPL	LGPL	Apache 2.0
Price	Starting $99 /mth	By consult	Free	Free	Free	Free	Free	Free	Free	Free	Free
Requirements	Browser (cloud)	100GB HDD, 2GB RAM	Unknown	Eclipse	Unknown	Unknown	10GB HDD, 2GB RAM	Unknown	Unknown	Unknown	Unknown
Category	Full System	Full System	Full System	Full-System	Engine	Engine	Engine	Module	Module	Engine	Engine
Documentation	None	None	Extensive	Extensive	Extensive	Extensive	Extensive	Very Sparse	Sparse	Extensive	Extensive
Background Tasks	No	No	Yes	Yes	No	Yes	No	No	Yes	Yes	Yes
Collaboration	No	Yes	No	Yes	No	No	Yes	Yes	Yes	Yes	Yes
Custom Tasks (such as Forms)	No	Yes	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes
Social Features (Chat, Helpdesk,)	Yes	Yes	No	No	No	No	No	No	No	No	No
Web Services	No	No	Yes	Yes	Yes	No	Yes	No	No	Yes	Yes
Web Based	Yes	Yes	No	No	No	No	No	No	No	No	Yes

2.5 Summary

In this chapter, we have reviewed existing solutions that we could use to fulfil our project requirements, namely engines for workflow management systems that we could reuse as a basis for the platform development. We have also study other related systems, like project managers and area-specific workflow managers that fulfil some parts of our requirements. We conducted a detailed comparison between these systems and have concluded that no solution completely fulfils our needs.

3 Requirements Specification

In the previous chapter, we did a detailed evaluation of most well-known workflow management/task management solutions already on the market. This knowledge of existing solutions is important to understand the expected result of this thesis, as well as some of the approaches chosen (from a requirement standpoint, but also from an architectural point of view).

3.1 Challenges

The *raison d'être* of this thesis, since there are already so many solutions on this field, is finding the sweet spot between the task manager and the workflow solution. This implies getting into an area that does not exist, in order to fulfil the documentation and collaborative needs of a more traditional task manager, but also catering to the needs of a proper well-structured workflow system, complete with Input/Output (IO) on tasks. The system needs to fill the needs of users that must repeat the same work patterns several times and with different collaborators, similar to the establishment of a protocol.

Other important challenge is keeping the system flexible and modular, so that in the future, the system can expand to include other functionalities, without having to destabilize the core of the system. Moreover, it should easily integrate with automatized systems that can rely on it just for documentation purposes.

Finally, and since there is already a lot of complexity associated with this kind of systems, it must be easy to use and understand, so that users without much background on this type of systems can easily use it.

3.2 Functional Requirements

With the previous challenges in mind, we will now specify a couple of essential functional requirements for the system, which are essential for the good operability and usability of the platform.

Create and Manage Workflows

We must be able to create and manage several workflows, which are composed of different types of complex tasks. These tasks have a relation between themselves, where they can have dependencies or pass content between each other through IO.

Repeatable Workflows

The platform should support repetition of workflow sequences across time, with different deadlines for tasks as well as different intervenient responsible for each task, on each run.

Duplicate Workflows

Users should be able to pick on other workflows and create duplicates where they can make changes without affecting the original workflow and without having to replicate it manually.

Reassign ability and Feedback

Users completing a given task can change in the middle of a workflow running process. We should be able to remove assigned users from the task. Task replying users can ask for feedback from the workflow overseer to get more details on a given task.

Private Workspace

Each user must be able to have a private workspace, where they can manage their running workflows and attributed tasks as well as all other personal aspects of the system.

Support document files

All users must be able to upload files related with both the workflows themselves and with the tasks, they are replying. Files should be interactive resources that user can update and comment.

Documentation approach

The system should document all actions. It should be possible to observe logs for all objects pertained in the system such as workflow runs, users, requests, resources, etc.

Notifications

The system should support notification in several key areas, using for example email, so users can receive feedback or requests, even when not authenticated on the system, and their feedback be pulled into the processes.

Form-based tasks

Users should be able to create small questionnaires to collect uniformed feedback on a given task across several intervenient and later analyse it to proper tools. This questionnaire answers should be exportable.

Platform administrators and approval system

The platform should support at least two different kinds of users, normal users and staff, the latter being responsible by managing the normal users of the system. Normal users must be able to register and wait for approval or rejection from staff users.

3.3 Non Functional Requirements

Multi-User

The system should support registered users, which can have their own workflow and tasks being able to collaborate through the system as independent entities to reach workflow conclusions. All users can perform the multiple roles of both workflow manager and task assignee.

Service-Oriented Architecture

All software must follow a service-oriented architecture, based on web services, so the system can be easily extendable to multiple platforms (as a client) such as mobile devices native interfaces. This also allows the system to integrate with other tools through automated means of accomplishing task conclusion, further down the path. The system must be able to run fully, without visual interface, running on web services.

Micro-Kernel Core Architecture

The core system must be concise and stable, and software extensions implemented through plugins, using a micro-kernel approach to achieve supportability by the system.

Easy deploy and platform agnostic

The system should be easy to deploy across any number of different platforms and environments, we need to achieve this while maintaining portability and only one base code.

Support mobile and modern web

With the ever increasing popularity of mobile devices such as smartphones and tablets, we came to a point where responsiveness requirement is a given on new platforms being developed.

The default client included in the system should be responsive and work well on small mobile devices, like for example cell phones and tablets. Support for desktop should focus on modern web practices and discard accommodation of old and outdated technologies, focusing instead on browsers such as Chrome, Firefox, Opera and IE > 10 that are future-proof and Hypertext Markup Language (HTML) 5 complaint.

The default client look should be appellative, but at the same time light, trying to reduce page load.

3.4 Proposed architecture

With all the previous specified requirements in mind, and for the sake of maintainability and scalability, we propose a detached two-layer architecture for a hybrid task-manager/workflow documentation system named “Taska”, in the following chapter. The proposal tries to address all initial requirements, with a web platform, while keeping the core of the system entirely service-based and independent, but not forgetting to make available a rich user experience through a client for this web services included by default with the system.

We can see below on Figure 2, the typical use-case diagram that resumes most of the functional requirements and separates all users in three typical roles. The researcher is the role of the user conducting workflow runs. The data custodian is the role of the user answering to tasks for the researcher. The administrator manages the users. Any user can represent at different points in time (or simultaneously) the mentioned roles.

Figure 2 - Use case diagram

Our hope, besides fulfilling the initial requirements of our EMIF initial use-case, is making available a platform which can be a reference for organizations with mixed documentation structured needs which are currently being solved using a mix coordination of manual processes and task managers such as the mentioned on the state-of-the-art chapter.

3.5 Summary

On this chapter, we have seen functional requirements describing what the system must be able to do, non-functional requirements describing how the system should work and a implementation proposal based upon the described requirements.

4 Web software patterns

In this chapter, we will describe several architectural software solutions and development frameworks that were evaluated before the development of the final proposal of this dissertation.

4.1 Software patterns

Software development is the process of transforming real world requirements into a functional software product [23]. Although still very young, when compared with other development areas, software development has been moving at a fast pace, as more and more of the real world processes become digital and completely reliant on software, which in turns increases the expected quality and performance for information systems, as well as the ability to rapidly change the requirements on the go [24].

The need for speed up the entire process of software development, as well as the need for a succinct common vocabulary, has driven software to naturally converge on well-defined patterns that are, by heuristic experimentation, recognized as the best solution in a given context [25].

As one could expect, we can have software pattern on different levels, ranging from the architecture itself, to the smaller structural patterns. In this section, we will take a look at the most well-known patterns that are currently used in most efficient and scalable software.

4.1.1 Layered Architecture

The layered architecture is the most common architectural pattern for software development. In fact, we can be using the layered architecture without even realizing it, since it is the pattern developers converge to, for being similar to the way humans naturally structure themselves and their work [26].

In a layered architecture, we laid software layers over each other, each layer handling a specific responsibility. Usually layered-based software is composed of four layers: presentation, business, persistence and database[26] (Figure 3).

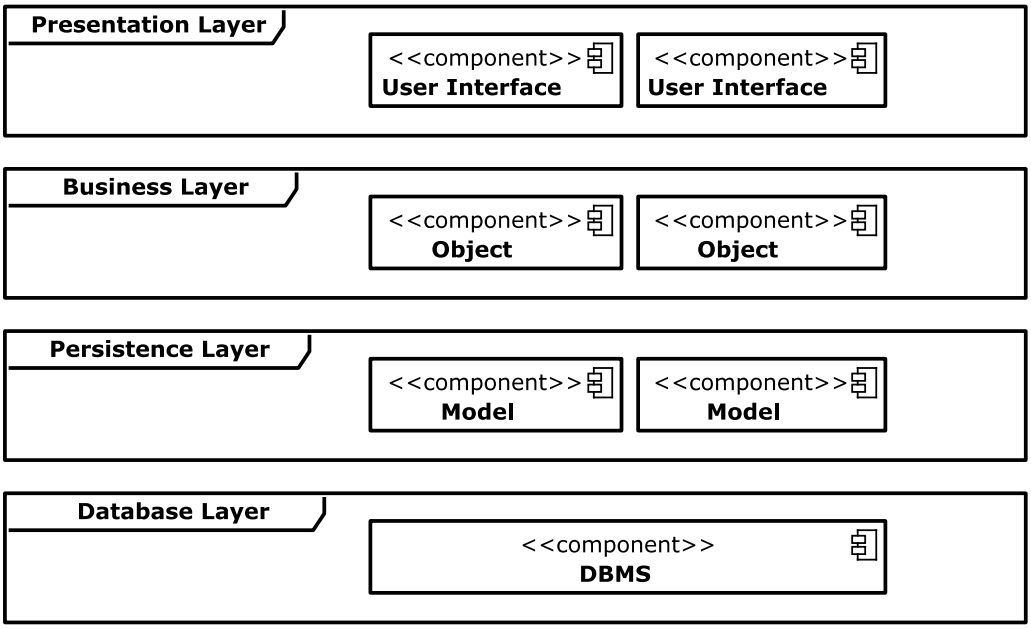

Figure 3 - Layered architecture example

One of the bigger advantages of the layered architecture is the separation of concerns, which makes it easier to develop and test. The layered nature does however, make deploys harder, since it usually involves a complete re-deploy and makes it harder for make structural changes if the layers are too tightly coupled.

We can conclude that the pattern is a good general-purpose pattern. Although not very extensible by itself, especially when the application is monolithic. The architecture also does not perform well, when there are too many layers, which makes it a bad fit for application that need high-performance levels (like data processing).

4.1.2 Microkernel Architecture

The microkernel architecture is a natural pattern to implement software that allows integration of a minimal application core, with a series of extra isolated pluggable features through a plugin system. The main concept behind the approximation is allowing extensibility and flexibility over a stable system, while providing complete isolation. Traditionally this architecture was implemented as a base for operating systems [26].

In this architecture, the system typically consists a core that only contains the basic functionality and the ability to know which plugins exist, and how to use them. All complex and extra functionalities are included through the plugin system (Figure 4).

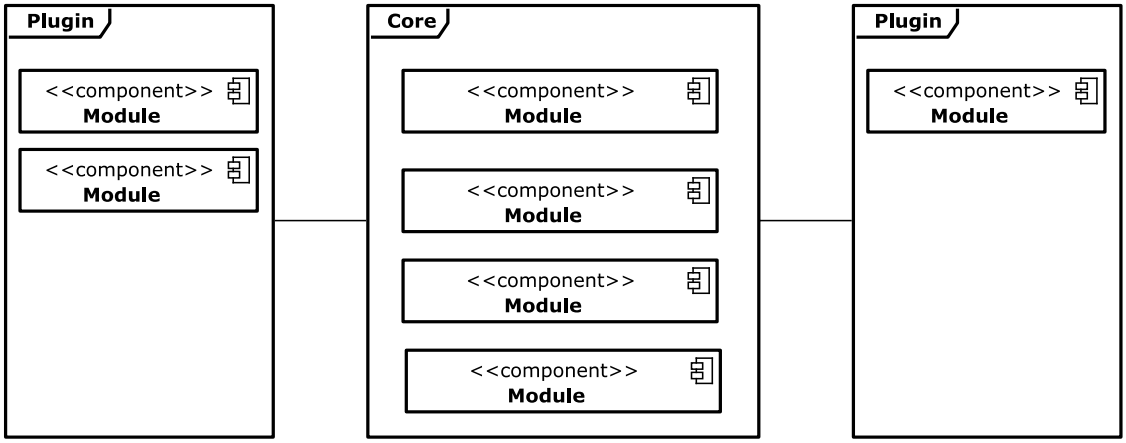

Figure 4 - Microkernel architecture

This pattern allows the development of systems very agile and easy to deploy, while allowing performance streamlining (since the systems don't have to include unnecessary features) [26].

We can conclude that this pattern is an excellent match to couple with the layered architecture pattern, complementing the lack of the pattern in some areas, like agility, extensibility and deploy difficulty.

4.1.3 Service Oriented Architecture

The Service Oriented Architecture (SOA) is a pattern for building distributed systems that makes functionality available as services to other applications or services through a accorded communication protocol [27].

In this architecture, the system has usually a layered structure, with the last layer being the service layer that works as the presentation layer typically would. This is however not mandatory as the pattern is agnostic, and we can have distributed system orchestration through an SOA architecture.

Below, we can see on Figure 5, an example of a service-oriented architecture.

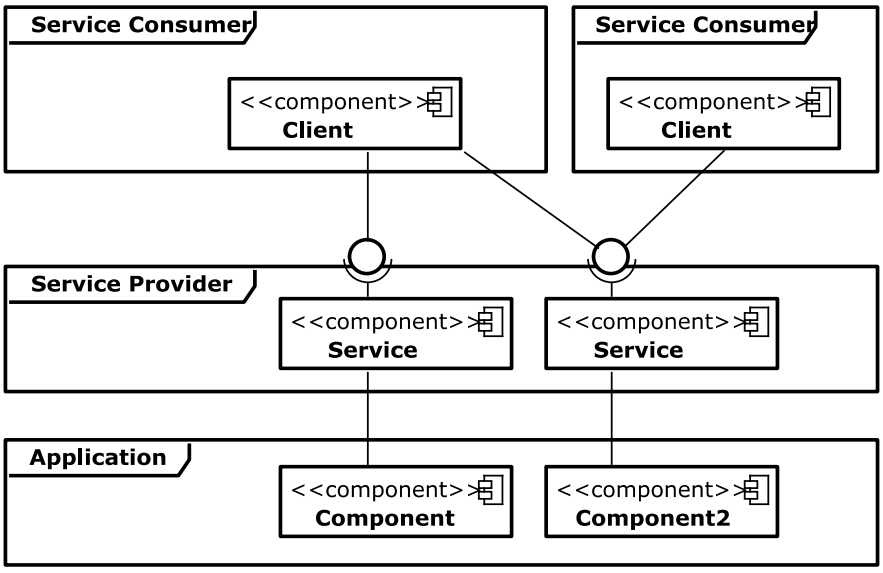

Figure 5 - Service-oriented architecture example

One of the biggest advantages of this architecture is the loosely coupled structure, which makes reusability and interoperability a given [27]. The most common usage of a SOA are web-services application program interfaces (API's), based on REST or SOAP.

One of the biggest disadvantages of SOA architectures is the fragility from changes on communication protocols. API changes can break client applications, so API's must be carefully crafted and made extensible trying all ways not to break compatibility.

4.1.4 Model-View-Controller Pattern

Model-View-Controller (MVC) is a best-known software pattern for implementing user interfaces and web applications, and is the basis of most modern web frameworks. The pattern separates the application in three components, the model that handles the data storage, the view, which takes the model and generates a representation for the client and the controller that updates the model state whenever there is an event. The main concept is separate the graphical representation from the internal structure [28]. Below on Figure 6, we can see an example of MVC architecture.

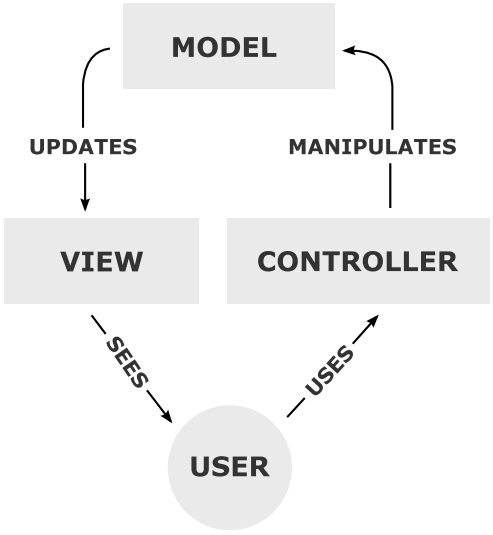

Figure 6 - Example of MVC architecture [29]

4.2 Modern Web Frameworks

A Web Application Framework is a set of components and tools structured in an organized pattern, to simplify software development of complex web applications. These frameworks reduce the amount of programming necessary by making common functionalities available and promoting code and concepts re-use [30].

Many web frameworks are currently available, implemented over several types of programming languages and over several software architecture patterns. The most common pattern on modern web frameworks is MVC, or some variant of it.

In the sections below, we will do a quick overview of some of the most used and known contemporary MVC web frameworks, discussing their capabilities, architecture, major drawbacks and advantages.

4.2.1 Server-side Frameworks

Java EE and JSF

The Java Enterprise Edition (J2EE) is a scalable platform to develop multi-tiered web applications in Java, and it is oriented to the development of large-scale enterprise applications. The platform includes a great variety of components, like the Java Persistence API (JPA) to handle the database abstraction and connection, the Java API for RESTful Web Services (JAX-RS) that allows the creation of REST web-services and Java Message Service (JMS) that allows software orchestration. The applications for Java EE usually really the representation of data in Enterprise Java Beans (EJB) to provide encapsulation [31]. Below, on Figure 7, we can see the typical Java EE architecture.

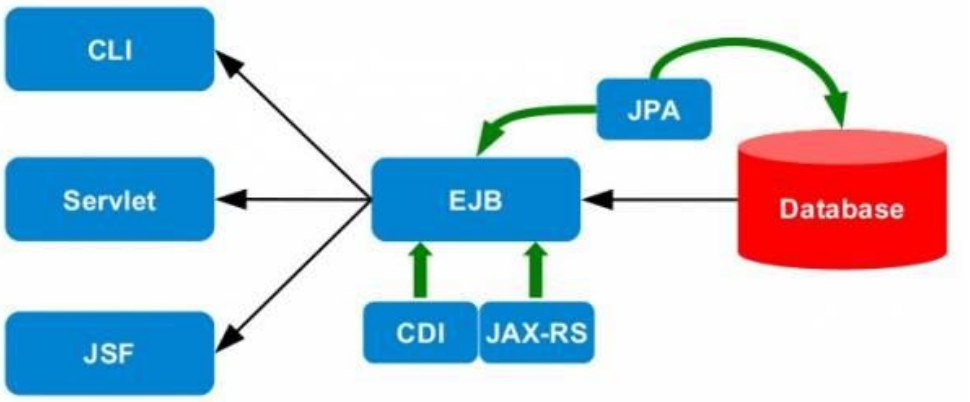

Figure 7 - Java EE typical architecture [32]

Java Server Faces (JSF) is a component, following the MVC framework standard that we can use over Java EE to provide data output and input to the client through xHTML pages. The default implementation of the JSF is called Mojarra [31]. In spite of working independently, JSF couples with other component frameworks like Primefaces or IceFaces that provide a series of complex ready to use components.

Java EE has a slow learning curve because of its complex structure, which requires intricate knowledge of the existing components. Even for an experienced developer, dominating the platform and all the facets of the development in order to take the best performance out of it takes a long time and effort. This does not make it a very attractive platform for projects looking for fast development cycles.

ASP.NET and Web Forms

Active Server Pages (ASP.NET) is an open-source web application framework, developed by Microsoft, over the .NET application framework. The .NET abstracts development language, allowing the development to be made using C#, C++, Visual basic, and others [33]. On Figure 8, we can see the generic ASP.NET architecture.

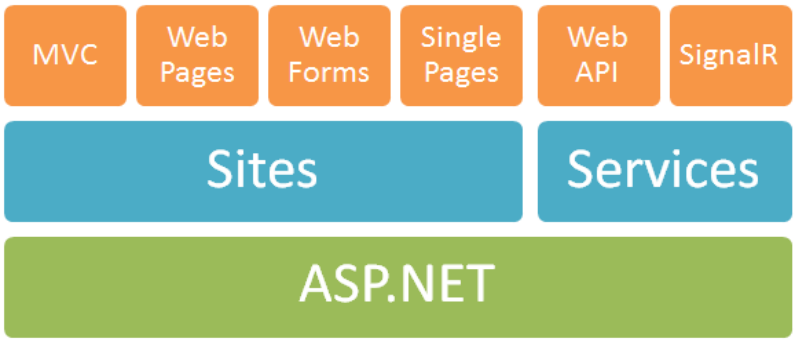

Figure 8 - ASP.NET architecture [34]

In Windows, ASP.NET has within Visual Studio one of the best Integrated Development Environment (IDE) and deploy environments currently existing. Developing Web Forms is very easy on Visual Studio. There is an excellent debugger and the editor even has a visual drag-and-drop editor to create the webpages layout [33].

The main problems with ASP.NET lay with the lack of web standards compliance (mainly on the client front), the tightly coupled architecture (which besides working as an advantage to make development easier, can also be a disadvantage in some cases when fine grained control is desired), and the lack of overall simplicity in handling the page life cycle. Besides, only running and being able to develop properly on Windows, which has license fees, makes it a bad option for open-source software development.

Play

Play is an open-source MVC based, lightweight, easy to use and highly scalable Java virtual machine (JVM) based web application framework that allows users to write code either in Java or Scala. Play applications do not require Java EE constraints using plain Java, but can run as Web application ARchive (WAR) files (exactly like Java EE projects). Play structure is very similar to Ruby-on-Rails and Django, as it was based on the former frameworks [35].

One of the main advantages of the Play Framework, when comparing with Java EE based frameworks, is the big reduce of complexity. The framework is very oriented to web developers, and does not try to cater to Java standards, but follows web standards instead, having really good modern Web support like HTML5, Web Sockets and other modern web concepts.

The biggest disadvantage is the relative young status of the framework, which makes it very volatile and subject to change. The documentation is very sparse to users used to Java detailed documentations. To further this problem, developers seem to don't care very much about maintaining backward compatibility between old and new versions, changing the API's in a way that make them incompatible between each other. This makes it a bad option for any complex project expecting to maintain long-term support.

Ruby-on-Rails

Ruby-on-Rails is a MVC open-source web application framework written in Ruby, designed to be sustainable and productive. Ruby-on-Rails is best known for making the Ruby programming language popular [36].

The main advantage of Ruby-on-Rails over some of the other web frameworks is the automation of boilerplate through naming conventions, which makes developing in the platform very easy. Rails has automatic imports, inferred template names and Object-relational mapping (ORM) for the database handling. The documentation is structured, and easy to use.

Below, on Figure 9, we can see an example of Ruby-on-Rails architecture.

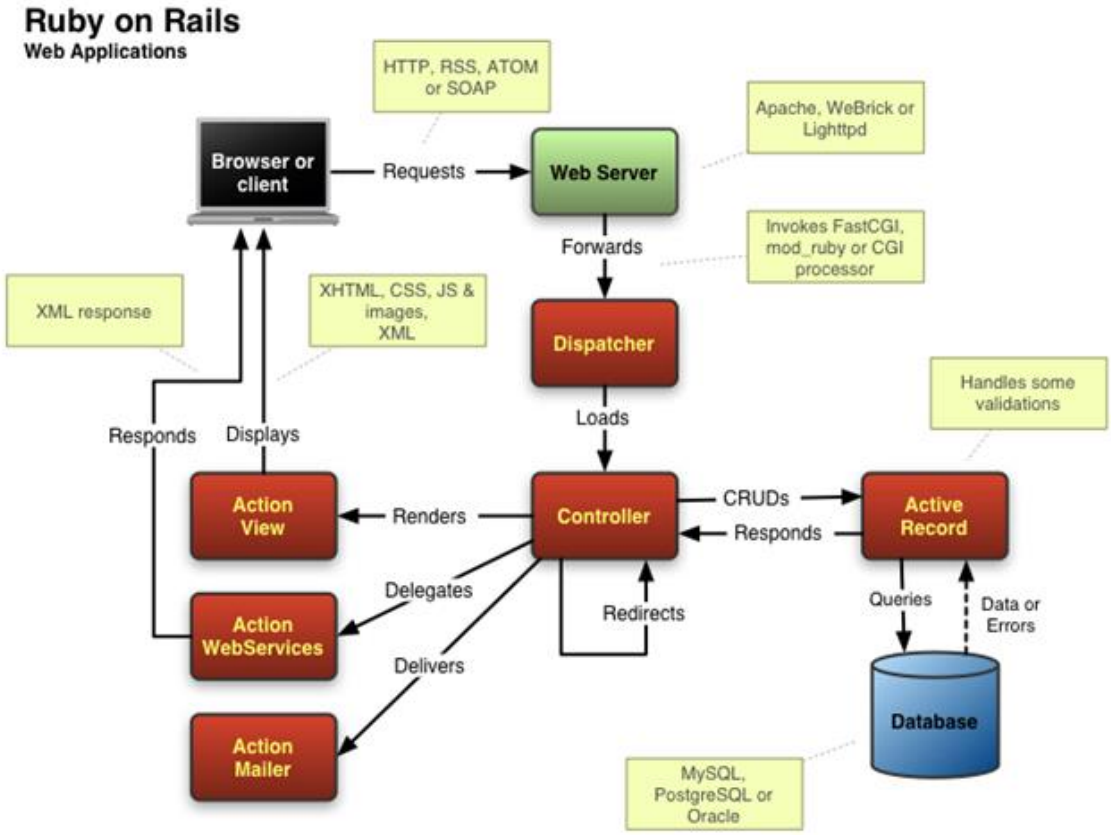

Figure 9 - Ruby-On-Rails example architecture [37]

The main problem with the platform are scalability and security issues, with big web services like Twitter having their code ported from Rails to Scala [38]. Rails also is very opinionated, fruit of complete automation, making it a bit inflexible, and hard to implement unconventional flows that do not obey the recommended pattern.

Django

Django is a MVC open-source web application framework written in Python, designed to encourage fast development cycles with a clean design, while maintaining security and scalability [39].

The microkernel architecture is the main concept for Django. A web application is just a bundle of applications, and there is a big community of dedicated developers creating and maintaining easily integrated third party applications. Each application is isolated and follows the MVC pattern.

Besides providing ORM, an automatic administrator interface, a good template system and a cache framework, the biggest advantage of the framework is the very extensive and detailed documentation provided and the care of not breaking

backward compatibilities. The framework has a big focus on support, which makes it a good choice for long-term projects.

On Figure 10, we can see an example of Django architecture.

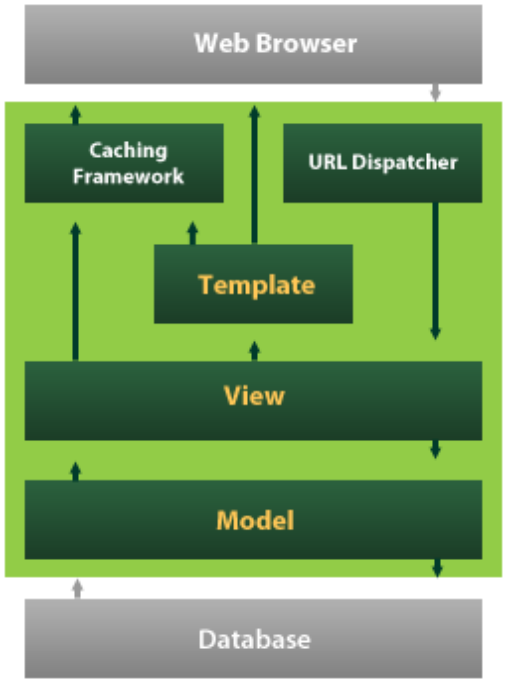

Figure 10 - Example of Django architecture [40]

The biggest disadvantage with Django, is the lack of good IDE's to develop in. Python IDE's do not have the same kind of integration Java IDE's like Eclipse have. And while there are some, even after having worked for a long time using python and Django, I could not find a better environment to work than a simple text editor, most IDE's are cumbersome and do not provide the shortcuts one is used to expect from other frameworks.

4.2.2 Client-side Frameworks

The standard language for modern dynamic web development for the browser is JavaScript. JavaScript, however, was not originally conceived with the complexity of today's requirements in mind.

The Internet, as a whole, has come a long way since its inception where HTML were just hyperlinked static documents [41]. Nowadays, Internet users expect pages they visit to be well designed and feature rich. The web moved from being essentially informational, to reaching the realm of full-fledged applications trying to look for interactivity. Effectively, most web-pages now have a behaviour more close to programs than readable documents, and that comes at the cost of increased complexity [42]. Browsers (as well as computers) are getting more powerful, and slowly the complexity is moving away from the server-side to the client-side.

Today, the web came to a point where the unorganized amalgamate of JavaScript files that update the page content itself on handle events, is not manageable or even productive. The usual web-page request and refresh cycle does not cut it anymore and there is a crescent need for organization on the page flow control to make it

completely dynamic. In this scenario, several web-frameworks that organize and optimize the page flow started appearing, and taking over the rich-web applications.

In the following sub-chapters, we describe some of the best current known client-side web frameworks for building structured web apps, how they work, their objective, and their advantages and flaws.

AngularJS

AngularJS is an MVC/"Model-View-Whatever" (MVW) open-source JavaScript framework developed by Google to create rich web-applications. AngularJS tries to provide a clear separation of concerns. The application follows the typical MVC pattern, having the view represent an instantaneous state of the model, and being updated by the controller which in turn updates the data model, that in sequence triggers an update on the view [43].

The main concept is proving an imperative way to manipulate the page, while maintaining the declarative structure of the template. We write the views in a template language based on HTML but that extended its vocabulary, allowing automatic two-way data binding with JavaScript objects and common programming functionalities like iteration cycles. We make the controller connection with views through scopes, declared on the template. Models are simple JavaScript objects [41].

On Figure 11, we can see an example of AngularJS architecture.

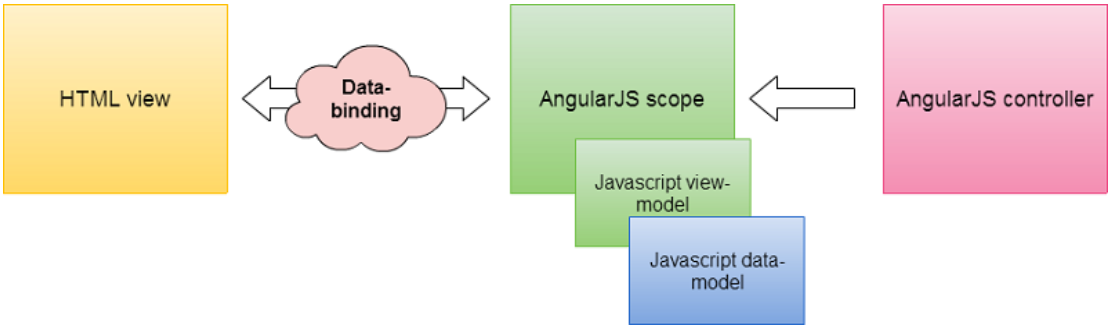

Figure 11 - AngularJS architecture [44]

AngularJS has a very big community of users with much more components based on it than most JavaScript frameworks. AngularJS also has a good documentation. The biggest problem with AngularJS seems to be integrating with non-angular based third-party libraries, partly because AngularJS highjacks the page events, which many (if not most) libraries rely upon.

Ember

Ember is a MVC open-source JavaScript framework for web application development built for productivity. The framework provides two-way data binding, a template engine called Handlebars, components to create custom HTML tags, and a complete platform for route handling [45].

Ember is based on the pattern of Convention over Configuration[1], inferring and automatizing as much as possible, like for example creating controllers and determining route names [46].

[1] Convention over Configuration is a software paradigm where the software follows a convention, and the developer only must specify the unconventional aspects of the application.

On Figure 12, we can see an example of Ember.js architecture.

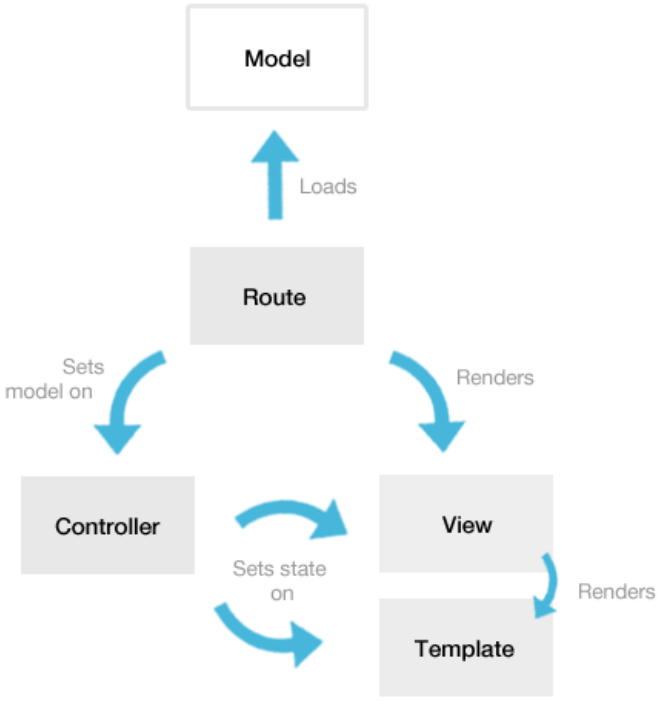

Figure 12 - Example of the Ember.js architecture [47]

While it may seem a good fit, one of the biggest issues with Ember.js is the instability of the API, breaking backward compatibilities very often, and debugging, which does not look well for a project hoping to maintain long-term support.

Backbone

Backbone is a lightweight open-source JavaScript framework that provides declarative event handling, models with key-value binding and is designed to integrate with RESTful JavaScript Object Notation (JSON) interfaces [48].

On Figure 13, we can see the typical flow for a Backbone application.

Figure 13 - Backbone typical flow [48]

The implementation is very simple, and for template engine, it relies on underscore template engine, which is a bit limited in functionalities and usually requires using mixed with JavaScript. There is also no two-way data binding, and views manipulate the page directly. The idea of backbone is not providing structure, but instead provide building blocks so users build their own structure, which is very un-opinionated, and can lead to lower production levels, or a lot of boilerplate.

ReactJS + Flux/RefluxJS

ReactJS is an open-source JavaScript library, developed by Facebook, for construct web user interfaces. ReactJS main core concept, relies on the existence of a virtual page, which can smartly update the real page, only for content changes

whenever it is needed [49]. Technically, on ReactJS there is no future page and past page, the page is always the same, which simplifies development efforts. React makes no assumption of other technology stack used in conjunction with it, and plays well with other kinds of libraries, like jQuery plugins, not designed with the virtual Document Object Model (DOM) principle in mind.

Flux is an open-source JavaScript library that implements the pattern of unidirectional data flow, a relative new architecture pattern that assumes data only flows in one directions, with no updates being done upstream, but always downstream, from the Store (data container) to the View [50]. We make content changes through Actions, which are queued and executed in order to prevent race conditions. We can use Flux to complement React capabilities. Facebook itself is currently using both of them in some of their web applications.

On Figure 14, we can see a diagram of a typical unidirectional dataflow architecture.

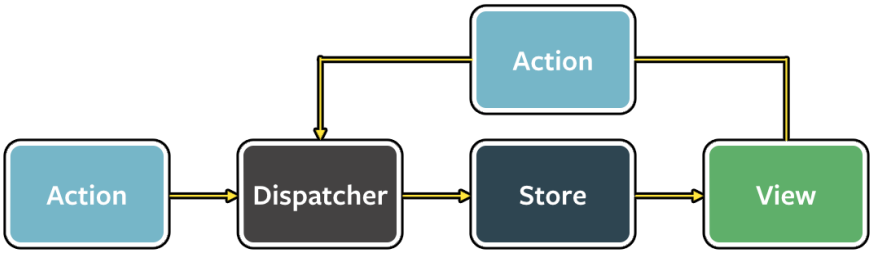

Figure 14 - Example of unidirectional dataflow architecture [50]

Together, and although Facebook rejects the strict concept, they form a structure close to MVC, with the Store providing the Model, the View being provided by ReactJS components, and the Controller being the Actions. The structure always presumes a dispatcher, which is responsible by handling all the action dispatches to the Store.

Reflux is an open-source JavaScript library inspired on the same architecture of unidirectional data flow as Flux, but takes a much more functional programming style, dismissing the Dispatcher, making all Actions a dispatcher and implementing the Observer pattern where stores listen in on Actions (or even other Stores) [51]. Reflux complements nicely with ReactJS, and can act as a simpler and easier to user placeholder for Flux.

On Figure 15, we can see the reflux application dataflow.

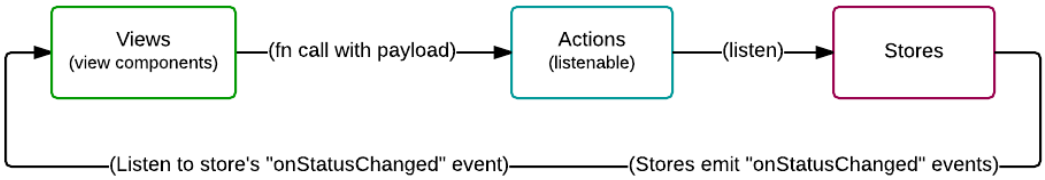

Figure 15 - Reflux dataflow [52]

The main advantage of the ReactJS and Reflux stack is immensely increasing the development simplicity, by making the component flat (since there is no need to care about the previous or next page state) and avoiding data inconsistencies that originate from race conditions via the unidirectional data flow pattern. More than that,

they are very un-opinionated and work well with already existing libraries like jQuery, jQuery-UI and other event-based libraries, as well as with modern JavaScript features.

The biggest problems are not having a routing solution (which is solvable via react components like react-router) and not having a default data fetching mechanism (which we can also solve easily since it is easy to integrate with jQuery to get the default jQuery JSON API). Other problem is the relative immaturity of the software, which is still in its infancy.

4.3 Summary

In this chapter we have seen software patterns useful for our platform implementation and a brief introduction to modern web frameworks that we could use for implementing our solution both server-side as well as client-side.

5 Taska – an extensible workflow management system

Following some good software patterns principles that we learned from the analysis performed in the previous chapter, we decided to develop the system core following a layered Software as a Service fashion, having a clear separation between the system core, based on web services, and its possible clients, based on whichever medium is deemed more appropriate.

In addition, to allow modularity and following a microkernel pattern fashion, we decided to allow plugging in new kinds of application, so we can add and remove new types of functionalities according to user needs. We used this kind of approximation on several key points of the core, ranging from types of tasks to the results.

We decided to make available a default client with web services. This way we leave the possibility of partial or full integration of other clients in the future. In this spirit, following the microkernel pattern, we also geared the default interface towards easy integration of new types of tasks.

In the Figure 16, we are able to see an eagle's eye generalized view of the overarching architecture components for the system environment.

In the sections below, we will take a detailed look at the technologies used to implement this approximation, from three different perspectives. From the backend engine core (Backend Core SaaS) perspective that secures the application business logic and works independently the others. From the frontend client perspective (Default Web Client) that relies upon the core web services to function and finally, from a deployment perspective and the efforts made to ensure effortless installation of new instances on all systems.

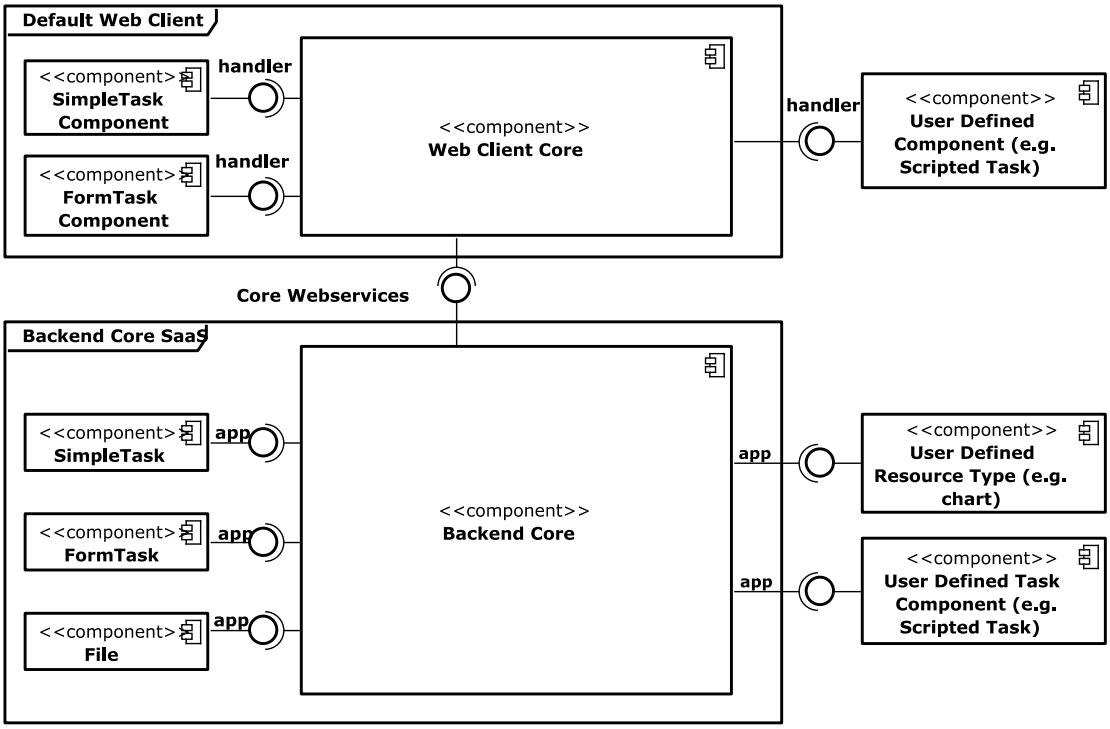

Figure 16 - Taska general architecture

5.1 Backend Engine

In this sub-chapter, we look at the technologies we have chosen to build Taska, how they connect to create the core functionality and to what services the core provides through the web services API.

5.1.1 Technologies used

Django

From the analysis of current state-of-the-art web frameworks, which can be seen the previous chapter, we decided to use the python based Django as the basis of the system. The main reason for the choice was the excellent documentation and support it provides, required to build a good core and stable system.

Other factor that heavily pended in Django favour was the low learning curve and the ability to have isolated and decoupled components in the form of applications, which greatly simplify the integration of a micro-kernel architecture, outlined as a requirement for our system to make it modular and expandable. Even better, requirements are a list of applications installed through package tools for python, such as Pip Installs Packages (PIP).

As previous analysed, Django has a big community of contributors which develop and maintain a very large set of third-party applications, which can be integrated in our application with little effort. As to not reinvent the wheel, we used some of these applications in our base core. Some of the more import applications used will be briefly presented below, and their usage explained

Django REST Framework

Described as flexible and powerful toolkit to develop Web APIs with ease, Django REST Framework is the de facto solution for building REST API's over Django. The

framework has a very well written and detailed documentation, much similar to Django itself and brings to the table usability features like a web browser-able API, support for oAuth and serialization mechanisms [53].

Since we are developing a SOA core for our platform's core, Django Rest Framework takes the vital role of communication point. We use it in all our modules, to provide the JSON web services we are providing to the exterior. In addition, provides a nice abstraction from having to process manually web services inputs and outputs.

Celery and RabbitMQ

Sometimes, web applications need to do processing tasks that cannot, or should not, execute during the normal request-response Hypertext Transfer Protocol (HTTP) cycle. Tasks like sending e-mails, aggregating big data or processing binary data in well-behaved systems that want to scale, should execute on the background, outside of the HTTP cycle.

Celery is a task queue based on distributed message passing with support for real-time or scheduled execution, where tasks can execute asynchronously on the background. Celery tries to be as generic as possible, so it relies on a message broker to handle the messaging, with support for several message brokers, like RabbitMQ, Redis, Beanstalk and MongoDB, CouchDB [54].

As per Celery own recommendations, RabbitMQ was chosen as the message broker, mainly because of stability and ease to install [54].

Celery supports direct integration with Django without any hassle, which made it the perfect fit for handling background tasks in our system, being them scheduled actions, or long-running events like sending notifications emails. More than that, Celery leaves open the possibility for implementing advanced functionalities in future versions of the platform that don't rely on user interaction, such as automated tasks.

Openpyxl

Python by itself has many functionalities. It can for example, generate CSV files without any third party library. However, nowadays, most users are used to handling excel spreadsheets, which have more advanced features and do not have the standard issue the CSV files have, with the value separator being different on different Operative Systems.

Openpyxl is a third party library to read and write Office Open XML files [55]. Users can use the library to export task results to Microsoft Excel Open XML Spreadsheet (XLSX) in our system.

Sphinx

Documentation should always be a big part of any well-structured project, since it makes easier to new users coming into the platform, as well as new developers to understand the project intricacies. Writing and maintaining documentation manually, however, is a very exhaustive chore, which can go out of scale fast. One solution for this problem is the documentation being, as much as possible, auto-generated.

Java has a good documentation system imbued on the language itself, Javadoc. Unfortunately, Python, the language Django is built upon, does not have by default, the same system. It does however has the third party Sphinx, which fulfils exactly these auto-documenting needs.

Sphinx is a very complete auto-documentation tool to make beautiful documentation. It has a syntax very similar to Javadoc that we can imbue on the code itself, allowing then exporting documentation in a variety of formats such as HTML, Latex, Portable Document Format (PDF) and Electronic Publishing (EPUB). The tool is so complete it's even used by Python itself to document the Python language [56].

In this project, we use it to document all important core functionalities.

PostgreSQL

Although Django has ORM, and the web framework automatically handles database interaction, as it does the database structure it uses, I thought relevant to mention the database driver we are using in Django is for PostgreSQL.

Although not the most popular, PostgreSQL is an open source stable Relational DataBase Management System (RDBMS) and the most advanced and standards compliant free Structured Query Language (SQL) system in the database market [57]. Django works very well with PostgreSQL, and using it is the approximation generally expected.

Sentry and Raven

One of the biggest time-consuming chores for any production-ready software is correcting existing bugs. Bugs are an inevitable part of any software and the way we handle them can make the difference between a software succeeding or failing in the real world. As such, strategies to bug logging are very important and should be part of any software architecture.

A very good solution for unified automatic bug logging across several deploys is Sentry. Sentry is a web application that provides mechanisms for logging and log analysing. It works virtually everywhere and although they business model is selling it as a service, the code is open-source, and everyone can make their own free deploy of Sentry. We have a Sentry deploy on "Instituto de Engenharia Electrónica e Telemática de Aveiro" (IEETA), which we use across several current projects.

The client for sentry integration for python is called Raven [58]. We use Raven, since our backend is written in Django, which is itself written in python.

5.1.2 Backend Architecture

Keeping in mind modularity and extendibility, we reduced the core of the application to the minimum possible, while thinking of the possibility of including new plugins for the system over time, through configuration only.

The system loosely follows the layered architecture pattern, mixed with the micro-kernel pattern for including new types of tasks, results and resources.

We accomplish the inclusion of new plugins through the default application mechanism of Django and we ensure interoperability by using Multi-Table Inheritance for key areas of the system.

Multi-Table Inheritance

Multi-Table inheritance (MTI) closely resembles the object-oriented notion of object inheritance. Instead of objects, we apply it to relational database models. In MTI, we have a common model for all inheritors, which extend this model. This translates in a common table for all common attributes and other tables with specific

attributes for each of the inheritors of the base model. This way, we can have a certain kind of polymorphism were the system can know and act upon the model instantiations without understanding or knowing about all the model details. In the Figure 17, we can see an example representation of MTI.

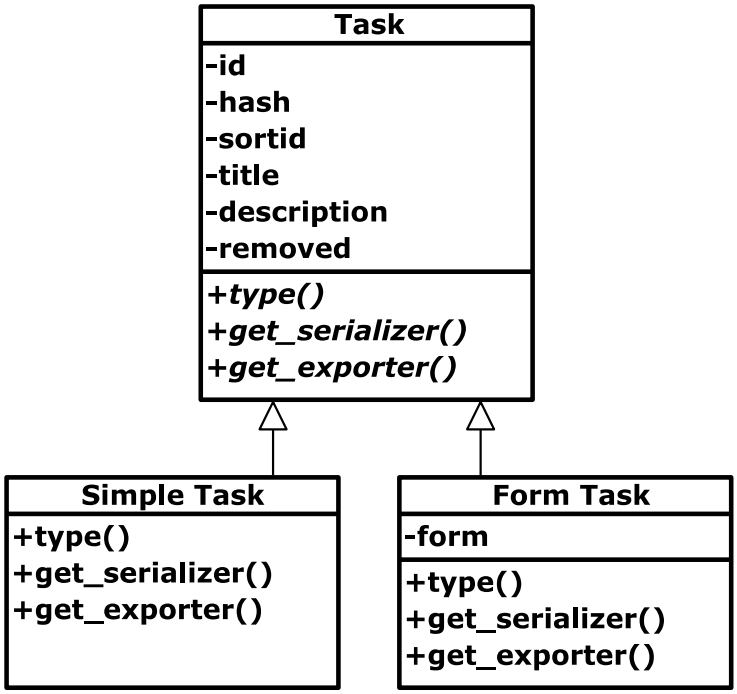

Figure 17 - Example of MTI

Django ORM supports by default MTI and we use it extensively in our platform to handle Tasks, Results and Resources, which can be polymorphic and take any form. With this, we couple a common interface of methods, allowing them to have different behaviours, but integrated views, without the system having to interpret them, and allowing better decoupling.

Identifications using hashes

In our system, we decided to use hashes instead of the unique ids as identifications for public objects. We have done it for two reasons. First, to allow unique identification further down the path, if we ever need to have a distributed database where monotonically increased values hold no meaning. Second, to obfuscate the number of objects existing on the system, and their order.

Backend Structure

Over the database persistence, which we maintain using Django ORM models, eight modules, separate from each other, were developed.

Overview

If we take into account all the modules, we came up with the following eagle's eye component diagram below that describes the core of TASKA backend, in an easy to understand way. We can see the layered architecture of the components, how they relate from a transversal point of view, and the modules providing services, as well as the modules accepting core extensions through new applications, without having to interact with the rest of the system.

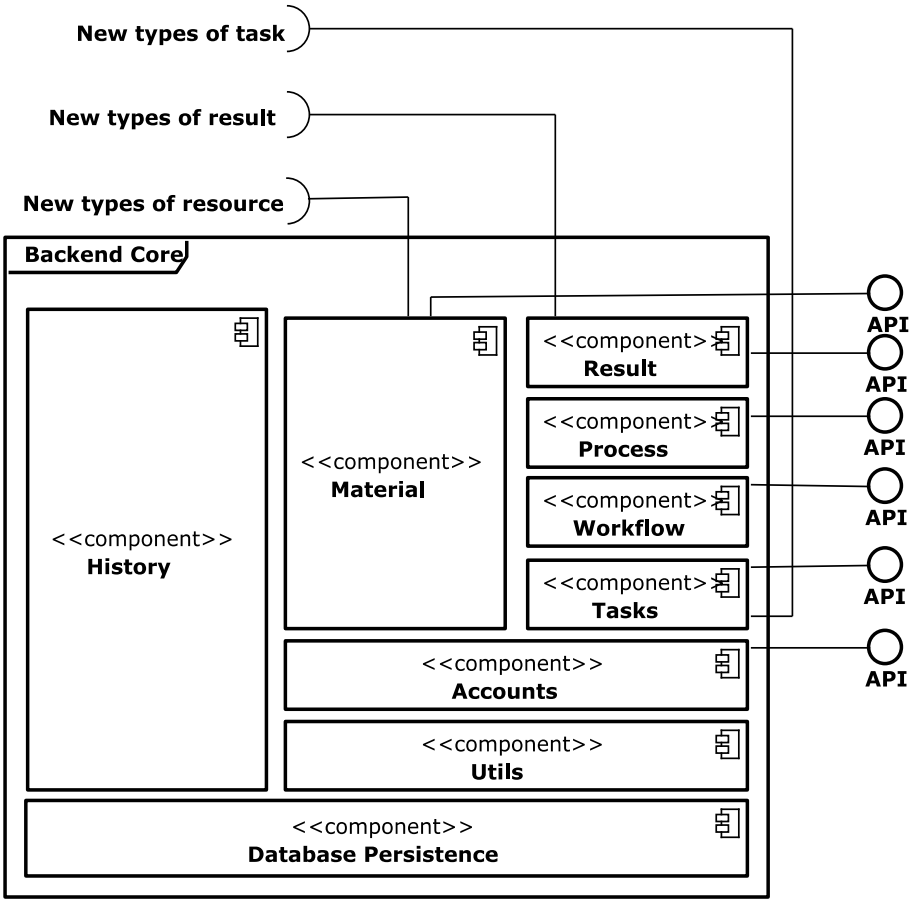

Figure 18 - Backend Architecture

Even though all the ORM is automatic with Django, sometimes it helps to be able to see an Entity Relationship Diagram, since it describes well how the system data is structured. On Figure 19, we show a prototype Entity Relationship Diagram (ERD) diagram, which while not describing perfectly the data structure, does allow to understand better the main data structures. This diagram, already includes the default plugin developed, the Form Tasks, which do not appear in the core backend architecture diagram, since they are not part of the core.

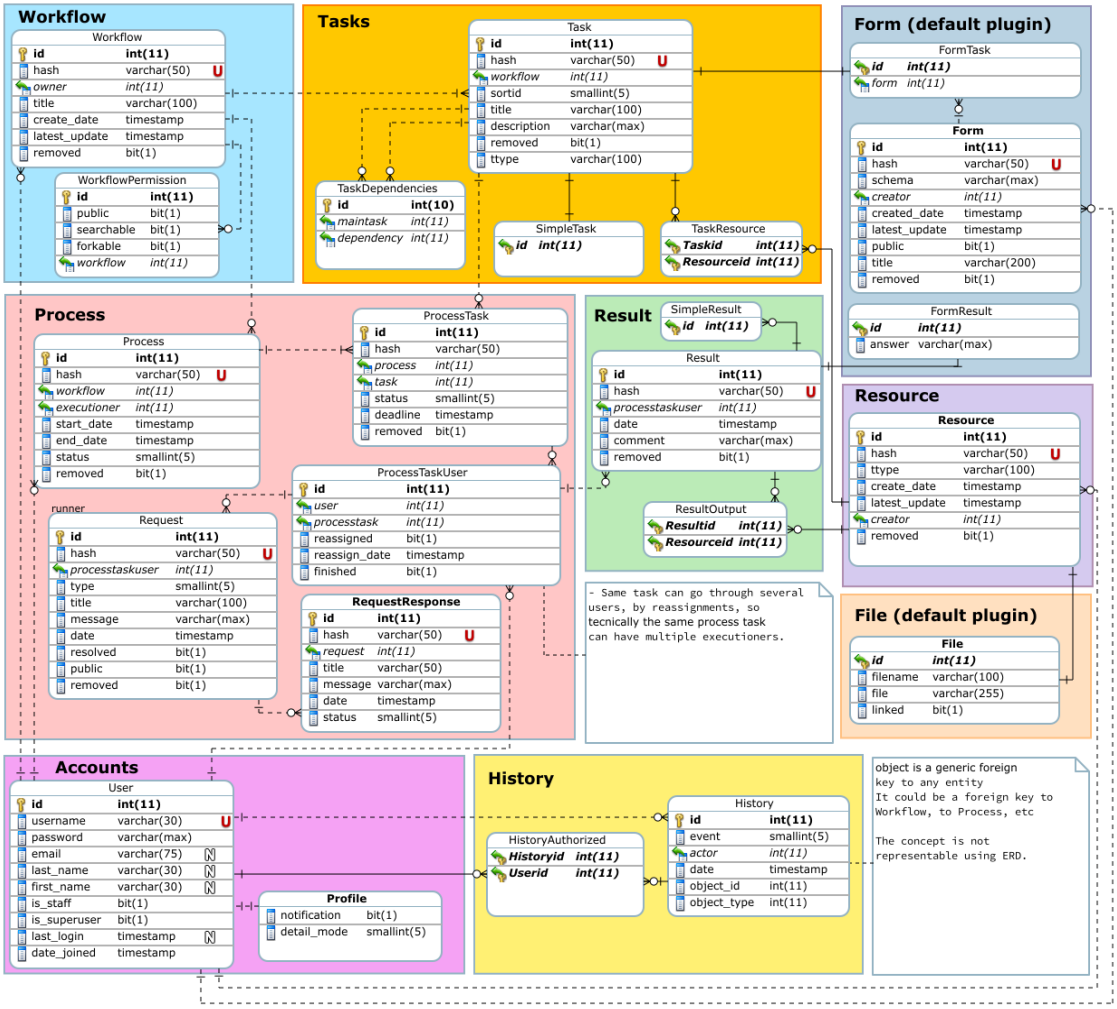

Figure 19 - Approximated ERD diagram

History

One of the first requirements for the system was to be documentation oriented. As such, one important part for a documentation system is the history log. In the platform, we try to record the history for everything. The history is in fact object based, and every public object on the platform has a history associated with it. The history component is transversal to all components and we use it to record historic events happening through the system. Every single object can record its events through the models History make available, and this is accomplished using a Django feature the framework itself is using to keep history on the Django Administrator automatic interface, the content types framework.

The content types framework

Sometimes objects relate themselves with each other in a complicated way. Relations between model instances is not always black and white and we can have models that intend to point to different models in its foreign keys. We would of course have different tables for each kind of connection, and then connect it together somehow. We could even achieve this through multi-table inheritance mechanisms like talked before. However, multi-table inheritance can be a heavy and more

expensive solution, especially when we do not need different behaviour, just generic connections.

The content type's framework implements a concept of Generic Foreign Keys, allowing models to have Foreign Keys that just point to any kind of model in the platform. We built the history component upon this concept.

History Model

We base the history model on the simple concept that all actions on the system can summarize into having an Actor, an Event and an Object.

The Actor is the User instance who makes the action. Someone triggers all actions with even automatic events having the user being the system.

The Event is a generic code picked from a list of possible events, made through the principle that we can resume and generalize all events into a finite list of possibilities. Events include Create, Read, Update and Delete (CRUD) operations, as well as complex operations (such as approval and commenting). This definition of an event code, allows us to filter events by type.

The Object is the instance upon where an Actor executes an Event. It is the Generic Foreign Key in our system and can point to an instance in any model in the system, even if the instance is outside of the core and in a separate plugin.

Besides these basic concepts, all history instances can have a list of authorized Users that can see it and a list of related Objects, which not being the Object in question, have been somehow related with the action that happened. An example of this, in context, would be events over a given Task related with the Study they are part off, the history is not for the Study, but is related. We accomplish this related relation through Generic Foreign Keys.

We can see the logic described on Figure 20.

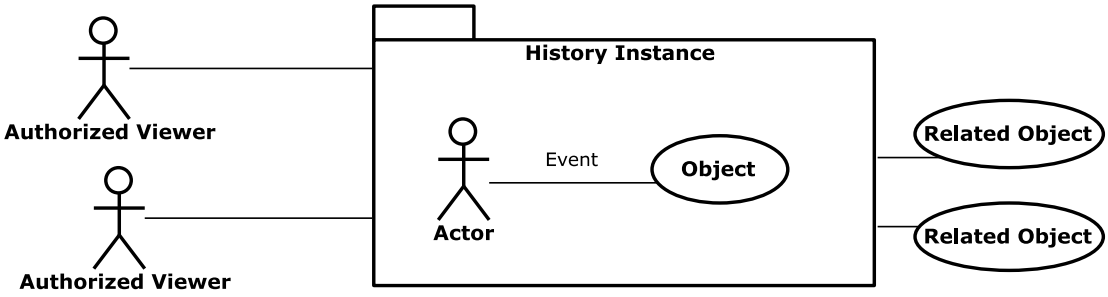

Figure 20 - History instance logic

One important aspect to keep in mind is history recording is active, not passive, so modules effectively call upon the history methods to record the history events. Even although we could achieve passive history recording, at least for basic CRUD operations through the Django signals framework. We do not do it so we do not record bogus information, but also because we intend to be able to record high-level operations further down the path, and gradually extend the list of event types.

Utils

One every project, there is always a need for a uniformed repository for boilerplate utility code needed throughout the platform core. The Utils module

handles assorted activities like, for example, those that generate hashes for other models and email content generation.

The notification system

As seen previously, history tries to document all actions on the system. This however by itself holds little meaning, if we do not try to take advantage of this notion and exploit it into our favour to simplify boilerplate activities.

The notification system means to inform users of certain events that pertain or relate to themselves. Since we know history is keeping record of all events, we can just infer that notifications will probably relate with events that we are already recording. Then, we came to the obvious conclusion that we do not need to develop code additional code for each notification, we can just use the history log events. Nevertheless, how do we catch them, without having to couple the applications? We use something called Signals.

Signals

As previously discussed, Django tries very hard to implement a kind of micro-kernel pattern. One of the principles of the micro-kernel approach is decoupling. The solution for communication between decoupled applications developed by Django is the signal dispatcher. Django allows applications to communicate through sending Signals, which can be caught anywhere on the platform. The good things about signals, we can make them available, even if we do not use them, just for new third party applications to consume.

With this notion in mind, we decided to make the History component send signals whenever there is new history, this way, we can have other application catch them. One such application is the notification system included in the Utils package. In Figure 21, we can see the flow of communication during a notification.

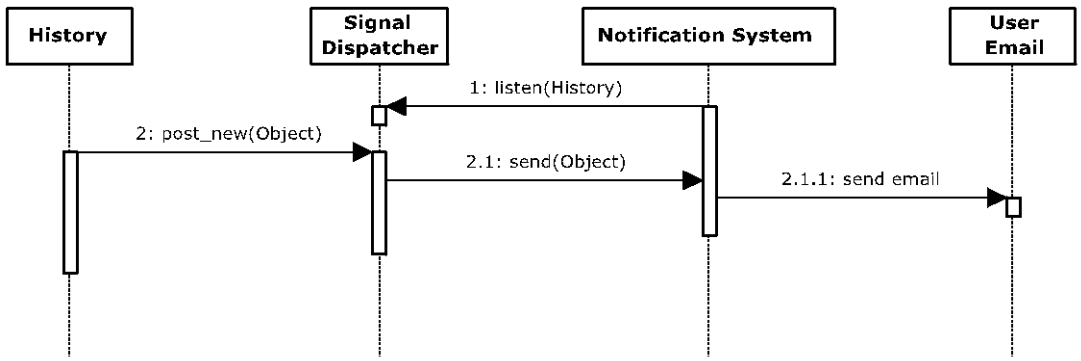

Figure 21 - Notification communication flow

"Easier to ask forgiveness than permission"

With the notification process pinned down, we still had the question of what we should notify, and how to create the notifications and send them to the users.

The typical medium for notifications is the email. For simplicity sake, we decided to implement notifications through this medium only, at least for the system initial version. Users can already see the history log, so it is much more interesting notify them by email, for when they did not see it on the system.

The other problem is what content we should notify. Since we are potentially catching all history events, how should we determine what is notified in a dynamic

way? In regards to this, we decided for an HTML template system that follows the typical Python approach. It is better to ask forgiveness than permission, so we presume everything is a notification, and we try to look for a template for it.

The template follows a convention over configuration approach. Each template should have a *<model_name>_<event_short_name>.*html and a *<model_name>_<event_short_name>_subject.html*. Text only versions of this templates are automatically generated using the nice html2text tool, which allows HTML to readable American Standard Code for Information Interchange (ASCII) conversion [59]. Whenever there is no template we can find, we just do not send the notification.

Accounts

The accounts module is responsible by the user handling in the core system. Since Django handles by default most of the user authentication and details, we build this module around providing the services through the uniform API developed, and keeping the User personal preferences and settings. Here we keep settings like if the user is interested in notifications, or any other setting that is relevant and appears over time.

Usually all objects on the system, being them Workflows, or processes or task instances, belong to a user, and interact with this to get their details.

Material

Documents are always an important part of any task manager. Usually tasks end up generating a series of documents or other resources as a product of related work. Even if they do not generate input, they usually will have input documents related to concluding a task. In our hybrid system, we also decided to give them special attention.

In this spirit, we tried to allow documents on the system to be an integral part of all processes. More than that: documents are implemented as generic resources through MTI, so even although for the basic system we just support File resources, we will be eventually able to cater to other types of more dynamic resources without having to change our core. Resources are so important that process intervenient can comment upon resources, and discuss them directly.

Dependencies receive resources, and we pass down these resources to dependants through the system. The resources end up being effectively the vehicle of communication for the task executioners in the system.

Tasks

Tasks are the basic unit of the system, and as such held in high regard. Using the MTI approach, we designed the Task and their web services in a way that we can easily integrate new types of Tasks, by including new applications on the platform.

While the Task does not directly connect to the Processes, they are composed using a Workflow, which works as a repeatable template for processes to be run, and instantiated as needed.

Tasks have inputs and outputs, given from the combination of their composition through the Workflow template dependencies, and their instantiation through a process where the task is attributed to users. Users can export the task results into a number of formats, like JSON, CSV and XLSX.

By default, the system includes two types of tasks: Simple tasks and Form tasks. We implemented Form tasks as a plugin for the system core, and as such, we will detail it in a separate chapter below.

At the base, a Task must fulfil a few requisites. First, the Task must extend the default base Task model, this is the absolute requisite for it to work, since this inheritance is the way MTI connects everything.

Then, the Task should implement its own content Serializer for the web services. We provide a default TaskSerializer, which can understand all the common fields. For all other new functionalities, the model should implement its own Serializer that extends the default Serializer, which follows the Serializer class specifications of Django-Rest Framework.

The Task models are responsible for the implementation of both Serialization and process task exporting, with a default implementation provided, but we expect a specific implementation for complex use cases. The exporter should extend the default ResultExporter model, and at least provide the export method that should return a list of rows, which will be included in the export format.

On Figure 22, we can see a schematic vision of a new extension implementation.

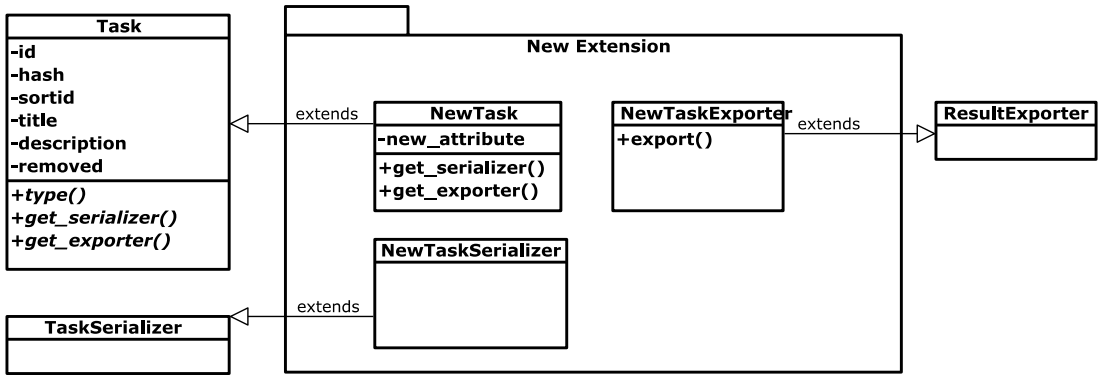

Figure 22 - Backend core extension implementation

Workflow

A Workflow is a composition of dependency related tasks that can be of any type, structured in an ordered way to accomplish objectives or results. The purpose of workflows is specifying a structure, which mixes tasks with the workflow of the tasks. This workflow can then be repeatedly executed as independent processes, with different participants and deadlines. We decided that for now, the system should follow a state machine, but without loops, since it would require a much more complex logic.

Although being owned by a user, a workflow also has a series of related permissions, like being public and being possible to duplicate them, which will work for even other users. It is very similar to a template or protocol by which the processes will run.

Following the previously mentioned business process example, we can see it modelled on Figure 23, in a similar fashion to what we could model in the system.

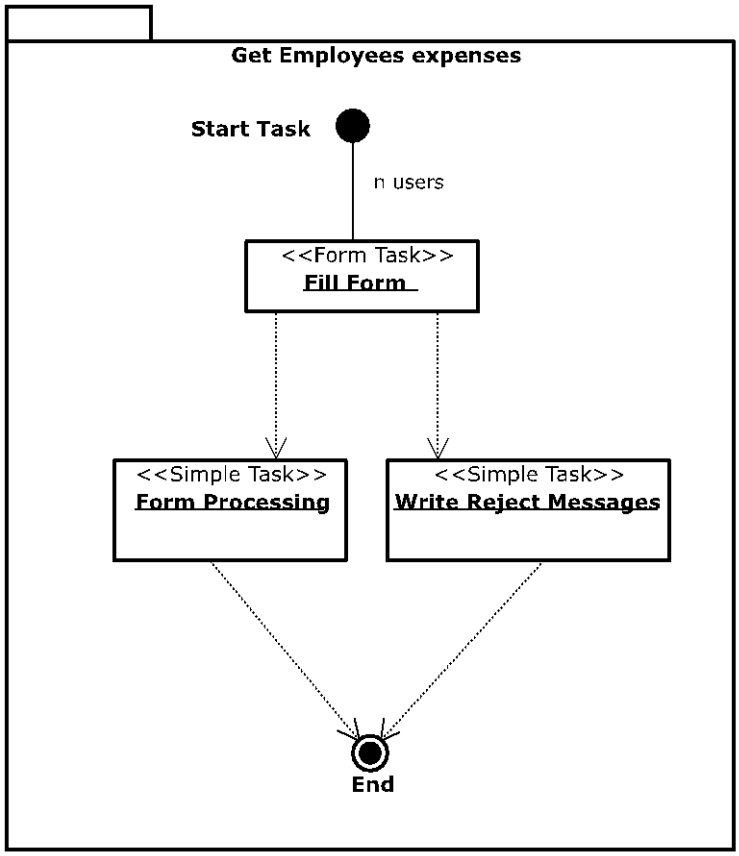

Figure 23 - Example of a workflow

Process

A Process is an instantiation of a Workflow with a specified group of users and deadlines. To the act of instantiating a Process from a Workflow, we call it a "run". When running a process, we have a series of states: running, finished, cancelled or overdue.

Whenever a workflow is instantiated, all its tasks are also instantiated into something called process tasks. Tasks also have a series of possible status: waiting, running, finished, cancelled and overdue.

Process tasks link Processes and the Tasks from the workflow template and can have multiple users, which are also instantiated at the run time, but we can be add during the process, for tasks still running or waiting.

At instantiation time, the process looks at all the process tasks instantiated and looks for dependencies. If the process task has no dependencies, the state changes to running and users receive their tasks. All other tasks with dependencies remain waiting. Whenever a Task is completed or cancelled, there is a re-evaluation of all tasks still waiting, and as we fulfil dependencies, the path moves along. When a task has multiple participants, all users must finish the task, or the user running the process must cancel the task, otherwise the system will not move along. We can see this process in a much-resumed form in the activity diagram of Figure 24.

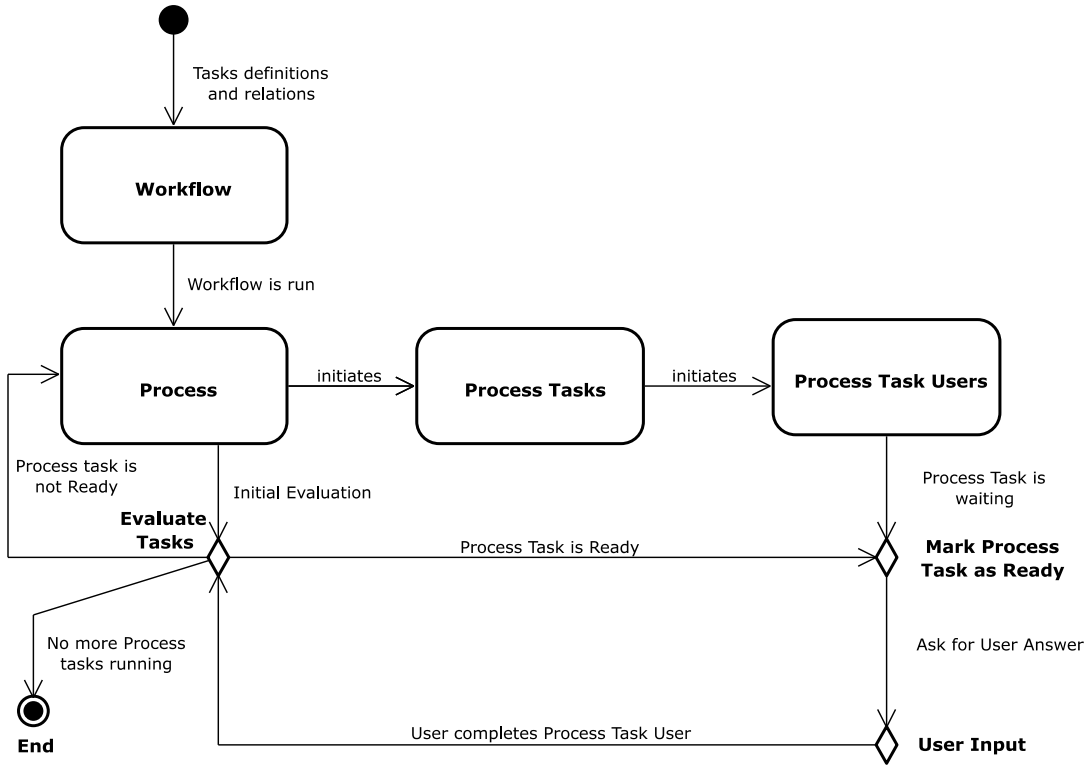

Figure 24 - Process state machine

All workflow management tools eventually need a medium of communication between the user completing the task and the user overseeing the workflow process. Another facet of the process module is the handling of the requests.

Requests on the platform are very similar to a messaging system, by which the users performing tasks can ask through small messages for clarification or reassignments of tasks. These requests notify the process overseer, and the user replying to the task can consult their responses. Besides all of this, the process overseer also has the ability to make this requests and responses public and available for all users completing the same task, that way mitigating possible duplicate answers for the same task.

Result

All Process Tasks Users must inevitably reply to the Process Task, with a Result, unless of course, the overseer cancels their participation or the process. Therefore, this module is responsible for the handling of the user responses.

Just like in the Tasks module, and in light of wanting easy extensibility, all Results implement the concept of MTI, since if we have different kinds of Tasks, we will probably have as many different kinds of Results.

In a way similar to the creation of new types of Tasks, when creating a new type of Result, one most fulfil mandatory requisites, for them to be available on the system, being the main one extending the default Result model, from the core system.

After this model extension, and if the result needs functionalities the default ResultSerializer does not provide, the Result should also specify a content Serializer to be used by the web services. Such as the TaskSerializer, a module that closely

follows the Serializer class specifications of Django-Rest-Framework. A schematic for this result extension can be seen on Figure 25.

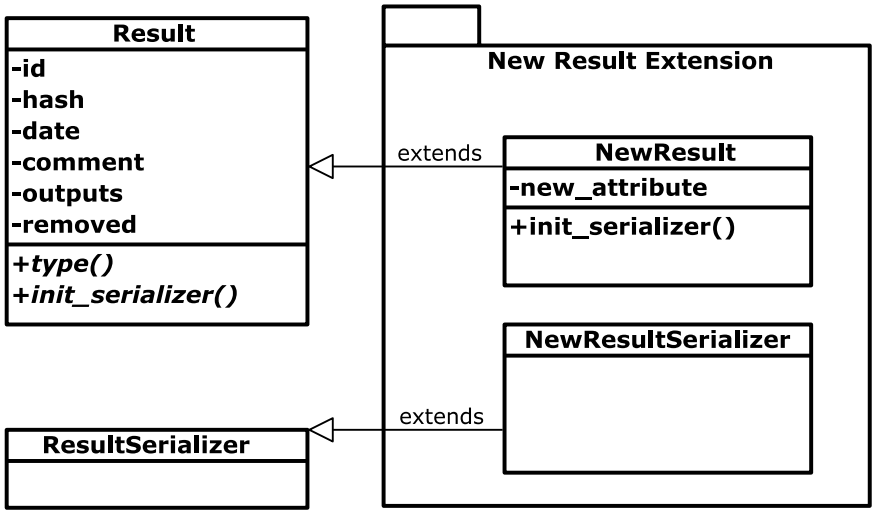

Figure 25 - Result extension

We will now take a view of all the web services made available through the exposed components we have seen on Figure 18 (indicated by the API connector).

5.1.3 Web Services

All web services are mainly JSON based, excluding obviously file uploads which are handled as binary data. In this section, we provide a brief explanation of all the web services that compose the backend of the workflow architecture.

For each method, we can also see the structure of inputs and outputs on the respective annex section. The correspondent numbers are marked on each entry on the table. Listing web services have pagination which follows the GET parameter *'?page=<x>'* and *'?page_size=<x>'*. They also have ordering with the get parameter *'?ordering=<field>',* and filtering by using *'?<field>=<value>'*. For each listing web services, we also list the possible filters and order fields in the annex.

All web services start with the path *'/api'*, after which follows one of the main areas, which then have sub paths specified in each table entry.

For additional information on all web services, this can be obtained by using the path of the web service with the method OPTIONS. Web services of type PATCH, allow use to only update some of the fields, there is no need to pass them all.

Below, on table 2 through 9 we can see the available web services.

/account

Staff only

Path	Method	Description	Annex
/	GET	Return a List of Users	9.1.1
/	POST	Create a new User	9.1.2
/<hash>/	GET	Returns information about a user	9.1.3
/<hash>/	PATCH	Updates information on a user	9.1.4
/<hash>/	DELETE	Logical deletes a user	9.1.5

/activate/	POST	Activates an inactive user	9.1.6

Table 2 - Staff Only Web Services

All users

Path	Method	Description	Annex
/login/	POST	Logs in an user doing case insensitive email validation	9.1.7
/logout/	GET	Logs out a logged in user	9.1.8
/me/	GET	Returns session personal user information	9.1.9
/me/	PATCH	Updates session personal user information	9.1.10
/check_email/	POST	Checks if an email is available to be registered	9.1.11
/register/	POST	Allows users to register themselves. Being then put on a waiting list to be approved.	9.1.12
/recover/	POST	Allows users to ask for password recovery (which needs to be confirmed).	9.1.13
/changepassword/	POST	Allows users to change their own password, after confirming a password recovery.	9.1.14

Table 3 - User Web Services

/resource

Path	Method	Description	Annex
/	GET	Return a List of user-related Resources	9.2.1
/	POST	Create a new resource	9.2.2
/<hash>/	GET	Returns information about a resource	9.2.3
/<hash>/	PATCH	Updates information on a resource	9.2.4
/<hash>/	DELETE	Logical deletes a resource	9.2.5
/<hash>/download/	GET	Downloads a resource of type File, if the resource is of other type, no content is returned.	9.2.6
/<hash>/comment/	GET	Returns a list of all comments for this resource	9.2.7
/<hash>/comment/	POST	Adds a new comment on this resource	9.2.8
/my/upload/	POST	Uploads a file, creating a new resource of type File	9.2.9

Table 4 - Resource Web Services

/task

While the use of this API to create tasks is possible, we recommend managing the tasks through the workflow API, manipulating the Workflows nested tasks.

Path	Method	Description	Annex
/	GET	Return a List of user-related tasks	9.3.1
/	POST	Create a new task	9.3.2
/<hash>/	GET	Returns information about a task	9.3.3
/<hash>/	PATCH	Updates information on a task	9.3.4
/<hash>/	DELETE	Logical deletes a task	9.3.5

Table 5 - Task Web Services

/workflow

While it is possible to create all tasks previously using the task API and then connect them on the workflow, the workflow API allows the creation of complete workflows without ever need to use the task API.

We can do this through using imaginary identifications, called '*sid*', which will then be used during workflow creation to understand the relative position of each task and dependencies, and create needed new tasks. It is even possible to mix new tasks and already existing tasks with real hashes in this process.

Path	Method	Description	Annex
/	GET	Return a List of user-related or public workflows	9.4.1
/	POST	Create a new workflow	9.4.2
/<hash>/	GET	Returns information about a workflow	9.4.3
/<hash>/	PATCH	Updates information on a workflow	9.4.4
/<hash>/	DELETE	Logical deletes a workflow	9.4.5
/<hash>/fork/	GET	Duplicates a public or owned workflow, returning the duplicate	9.4.6

Table 6 - Workflow Web Services

/process

Path	Method	Description	Annex
/	GET	Return a List of user-related processes	9.5.1
/	POST	Create a new process	9.5.2
/<hash>/	GET	Returns information about a process	9.5.3
/<hash>/	PATCH	Updates information on a process	9.5.4
/<hash>/	DELETE	Logical deletes a process	9.5.6
/<hash>/cancel/	GET	Cancel a process, marking all current executing/waiting tasks as cancelled.	9.5.7
/<hash>/adduser/	POST	Add a new user to a process task that is executing or waiting execution.	9.5.8
/<hash>/canceluser/	POST	Cancel a user on an execution or waiting process	9.5.9
/<hash>/changedeadline/	POST	Change the deadline for a process task conclusion	9.5.10
/my/tasks/	GET	Returns a list of user attributed process tasks across all processes	9.5.11

/my/task/<hash>/	GET	Returns a specific user attributed process task, by id	9.5.12
/my/task/<hash>/dependencies/	GET	Returns the dependencies content for a specific attributed process task, by id	9.5.13
/processtask/<hash>/export/<mode>/	GET	Returns an export of all results for a process task in a defined format, such as JSON, XLSX, or CSV	9.5.13

Table 7 - Process Web Services

/result

Path	Method	Description	Annex
/	GET	Return a List of user-related results	9.6.1
/	POST	Create a new result	9.6.2
/<hash>/	GET	Returns information about a result	9.6.3
/<hash>/	PATCH	Updates information on a result	9.6.4
/<hash>/	DELETE	Logical deletes a result	9.6.5

Table 8 - Result Web Services

/history

Path	Method	Description	Annex
/	GET	Return a List of user-related history	9.7.1
/<model>/<pk>/	GET	Return a list of user-related history, for a given object	9.7.2

Table 9 - History Web Services

5.2 Frontend Client

In this sub-chapter, we will explain in detail all technologies used for building the frontend default client that consume the backend web services and displays the information in a nice and easy-to-use interface, which makes available workflow management by visual interaction.

5.2.1 Technologies used

ReactJS + RefluxJS

Software development benefits largely from organization, and reuse. Frameworks, solve well-known problems, in a well-known way, as such we obviously want to use some in our client.

For the main JavaScript framework to use on the development of the default client, we decided to use the state-of-the-art ReactJS, which already has been analysed in this document. The main factors weighing on this choice were several. The nice integration with the unique data flow pattern, which seemed a good fit for the platform we were intending to implement. The capacity of allowing faster development cycles. The active community which is continuously working on improving it and last but not least the easiness to integrate with existing JavaScript plugins based on jQuery, given the unobstructed nature of the framework.

In combination with ReactJS we decided to use Reflux, instead of the typically paired Flux. This decision has more to do with Reflux being an alternative more functional programming oriented implementation of the unidirectional data paradigm,

which does not require us to have a dispatcher, relying instead on the observer pattern.

Bootstrap

Although choosing a JavaScript framework goes a long way into improving frontend development speed, a big part of web development is layout structure and appearance definition. Definition of a webpage layout is a very time consuming activity, so a good idea for fast software development is looking for ways to remove this waste of time.

Bootstrap is currently the most popular open-source key-in-hand solution framework for web development of layouts [60]. It features a great community of developers, is mobile friendly and offers all kinds of interface gimmicks someone usually needs when developing a webpage, such as buttons, navigation bars, breadcrumbs, modals, panels and much more. We have chosen bootstrap as the layout framework for the platform client.

Font Awesome

When we take a good look at visual interfaces, we will notice one thing: most interfaces have many icons. Icons are a fundamental part of most modern user interfaces and the main reason, besides making the interfacing pretty, is allowing users to identify what each thing is.

Currently one of the most well-known free open-source toolkits for web icons is Font Awesome [61]. Font Awesome offers scalable font-based vector icons that scale well, and are lightweight requiring no JavaScript or images. In our platform, we use this toolkit intensively.

JQuery and jQuery UI

JQuery is a lightweight and feature-rich JavaScript open-source library which harmonizes JavaScript functionality across all browsers, making manipulating DOM very easy [62]. Now, jQuery is foundational, and most web pages use it to make code cleaner and smaller while avoiding cross-browser issues. Although we do not need it to make DOM alterations since we are using the unidirectional data flow, we still use it whenever we need to retrieve DOM elements.

JQuery UI is built upon jQuery functionalities to provide interface capabilities such as drag, drop, resize, select and sort [63]. Since we had do develop a visual editor for our workflow state machines, it was obvious we would need to handle user interaction such as drag-and-dropping elements, so we decided to use jQuery UI for this job.

Dobtco Formbuilder and FormRenderer

We decided to develop our own visual state machine editor since we could not find a good enough open-source solution for this task written in JavaScript. We could however find a very good solution for the visual creation of our forms, and that solution was Dobtco Formbuilder [64] and FormRenderer [65]. These nice tools allow us to create form schemas on the fly, using drag and drop, export them and then render them to obtain the results. Since there is no advantage to rebuilding the wheel, we decided to integrate it as part of our Form Tasks, although we changed them a bit, to conform to our style and way of seeing the questionnaires.

MomentJS

Handling dates is something every application must do. We must compare them, checking time difference, getting relative dates to current date. We accomplish this, and much more in JavaScript with the open-source MomentJS.

MomentJS is a very good date and time plugin [66], which allows us to easily manipulate dates, without ever having to interact with the native Date JavaScript implementation and we use it extensively on the default client.

React-router

One big part of web pages that for most people can go unnoticed or dismissed is the URL. The URL is the most important concept on a webpage, since it is the path used to access a content. Inside a webpage, there must be a mechanism that handles the URL processing and page retrieving. We call this mechanism routing.

Since we are using a decoupled architecture, besides the routing for the backend, our client side application must also have a routing mechanism of its own. The mechanism we used was React-Router. React-Router is a routing solution designed for react.js that integrates seamlessly [67] and provides support for nesting and transitions.

Currently react-router seems the de facto solution for routing when using ReactJS together with a SOA, so we decided to use it for our routing needs.

React components

Although still very young, React structure naturally incentives reuse, and as such it is completely natural components keep popping up, removing gradually the need to develop some common functionalities. In our project, we used some already existing components.

Griddle-React is a non-opinionated, highly configurable grid/table react component [68]. It features nice functionalities such as infinite scrolling, pagination and filtering, while not forcing any style on the table, which makes it easy to integrate with existing Cascading Style Sheets (CSS) frameworks like bootstrap. We use it for most tables on our client side application.

React-Breadcrumbs is a simple and automatic breadcrumbs generator component for react-router routing mechanism [69]. Since we are using react-router, we decided to use this plugin to be able to generate a navigation context, without having to write the breadcrumbs.

React-hotkey is a react component for handling hotkeys [70]. We use it to handle events on the state machine editor, like backspace and escape.

React-select is a flexible multi-select input react component, which supports Asynchronous JavaScript and XML (Ajax) and autocomplete [71]. We use this component whenever we need multiple selection, like when choosing users for a task in the state machine editor.

React-SimpleTabs is as the name suggests, a simple tab component built upon react [72]. We use this tab component whenever we need tabs inside a panel.

React-toggle is a good-looking react toggle component [73]. We use it instead of the native checkboxes, since the toggle is both more friendly and pretty on small screens such as mobile platforms.

React-widgets offers a series of react components to work as inputs, such as Calendar, Date time picker, and multiple select [74]. We use this mainly because of the calendar and date time pickers that are both great looking and user friendly.

Raven-js

As with Raven on the backend core, logging is also very important on the frontend side. Even more since as we feature a decoupled SOA architecture, the client code will actually be running on the user side, not in our server where we would have more control on how to log it. The solution of logging on Sentry through JavaScript is called Raven-js [75], and is used in our project for logging any occurring error.

Filedrop.js

File uploading is an important part of any web page, and as such, there is a large number of file upload applications. One of the best libraries, in terms of features, documentation and cross-browser support at the moment is Filedrop.js [76]. We use Filedrop.js in our default client application for handling the binary file uploads.

ECMAScript 6 and Babel

As with most programming languages, JavaScript is an ever-evolving language. The original purpose of the language is no longer valid and as time has gone by new requirements appeared. Today web development has been shifting from an exclusively server-based rendering approach to a distributed rendering approach backed by web services. As such, there has been a big focus on dynamic web browsers languages, namely the most well known and supported, JavaScript. Browser vendors, have been pushing JavaScript forward, but are always behind the specification of the language, as specifications tend to take time to implement.

Currently the most popular and well-supported specification of JavaScript is maintained by the European Computer Manufacturers Association (ECMA) and is called ECMAScript 5 (ES5). ES5 however is prototype oriented and lacks some basic object oriented concepts like classes, constants and modules, which makes writing organized code harder.

ECMASCRIPT 6 (ES6) solved these issues with their language new specification. This new specification brings classes, constants and modules into the table. Besides that, it also brings many constructs that are more advanced to the language, like arrow functions, generators, rest parameters and de-structuring. The only problem with this specification is that all browsers did not fully implement it yet and will not finish any time soon, probably taking some years.

It is in this situation that Babel is useful. Babel is a “transpiler” that turns ES6 into ES5 code [77] in a way we still can develop using the new functionalities of the specification, which are in the end converted to old ES5 specifications, which are supported everywhere, getting the best of both worlds. We developed all of our client code using ES6, for both speed of development, as well as organization and future proofing (when we reach a time Babel is no longer necessary).

NodeJS and NPM

One of the best features we can have whenever a developer is writing an application, is a package manager. Managing dependencies can be a real bother, more so whenever several developers are working upon a project at the same time. Package managers allow us to specify dependencies into a list of software names

plus versions, and in the end, everyone can retrieve all in the same way, through the package manager.

Traditionally, users do not use package managers in JavaScript environments, with most people putting the dependencies libraries together in a big bundle of files, which they then would include in the web page. This is however much unorganized and not scalable at all. Eventually tools started appearing to solve this problem, which worked like the tools already existing in other languages, like pip for python, and maven for Java.

One well known package manager for JavaScript is Node Package Manager (NPM) [78],which came from NodeJS [79], a JavaScript server application, but is independent and can be used without needing to write Node applications. Most well-known JavaScript libraries support NPM package manager and maintain a repository there. We use NPM as our package manager for our client-side application.

Browserify, Watchify and how everything fits together

We have discussed in what language we wrote our application, using modern ES6 syntax. We also have detailed how we handled our dependencies, using the NPM package manager. What we did not talk yet is how we make the dependency imports in the package environment and how we do the packaging and building process and most of all, how we make it fast and commode enough, so we can integrate the process into our development cycle.

Browserify adapts Node concept of importing modules through requiring them into the browser environment [80], it does so by constructing a dependency graph and packaging dependencies in the needed order into a single optimized bundle. The bundle can then be included in the html page in the usual way JavaScript files are included. We use Browserify to package our several application components into importable bundle at deployment time.

The problem with just using Browserify is that we have to run it every time we change the code, while this works fine for production files, at deploy time, this could make the iterative development process clunker than the usual. To solve this, a Browserify that watches file changes, and recompiles the bundle whenever needed appeared, which was called Watchify [81]. We use Watchify during our development process to generate bundles on the fly.

Incidentally, Browserify supports a variety of transforms, one of which is applying babel to the source files. This makes it possible for all the building and packaging of bundles be automatic, and instantaneous. Together they all form an amazing workflow, which while allowing us to work in an organized and easily reproducible environment, generating files that work everywhere.

UglifyJS and UglifyCSS

Whenever we think of web pages, one factor of huge relevance is how fast the web pages loads. One of the biggest enemies of fast page loading are huge JavaScript files and style sheets. It makes sense that one of our main concerns should be reducing script files sizes. One of the most common ways of doing it is “minification”.

“Minification” is the process of removing all unnecessary content from source code, effectively compressing it. Most JavaScript libraries have minified versions. Besides the bundle creation, we also minify our code, both JavaScript and style

sheets. For this task we used well known UglifyJS [82] for JavaScript, and UglifyCSS [83] for style sheets.

5.2.2 Frontend Architecture

As it was done with the backend core, we also keep modularity and extensibility in mind when designing the default client, following the same micro-kernel pattern for new types of tasks and results. Tasks are class based, and we accomplish inclusion of new types of tasks and results through class inheritance, and method overriding of abstract methods.

As was discussed previously, the React and Reflux combo, with their unidirectional dataflow, generate a three-tier structure composed of Actions, Store and View. As such, we construct all of the main components of our client applications having these three subcomponents. On Figure 26, we can see this client component architecture. For simplicity sake, in the diagrams that will follow, we will always represent them as a single component.

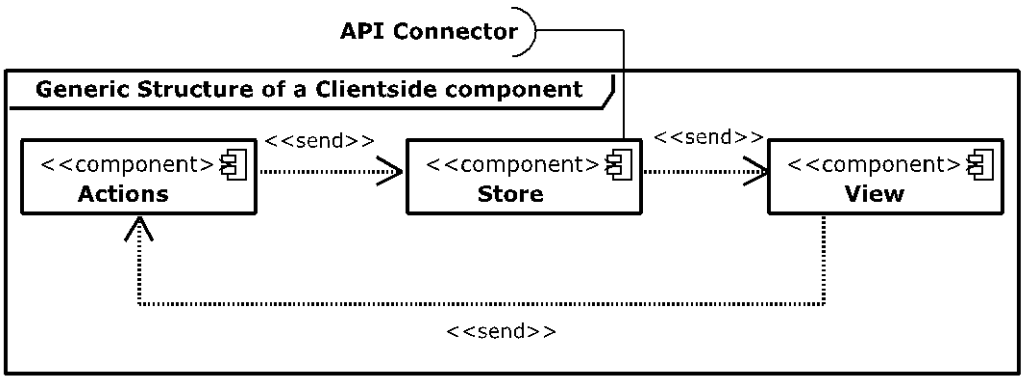

Figure 26 - Client component architecture

Generically, we have a major component for all the major concepts, which closely resembles the backend core structure, having components for User, Workflow, Process, Request, Task/Result and History. Besides these data-oriented components, we also have a component that handles API connections, a component that handles system state and finally two widget components developed to handle user interaction, which are state machine and the file uploader.

Overview

In the following diagram of Figure 27, we can see a resumed version of all architecture components and the way they communicate and use each other.

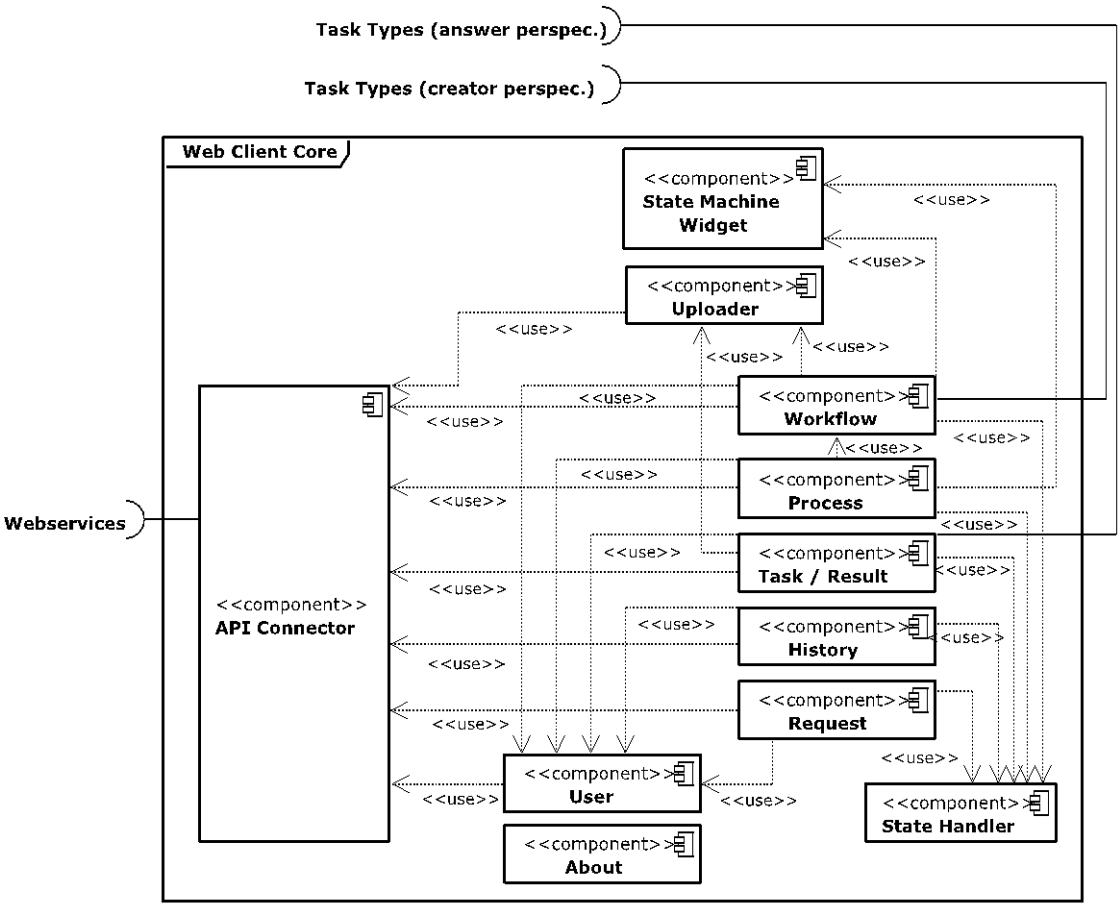

Figure 27 – Client-side architecture

We can see clearly, that almost all components end up using the API connector as the information source, and the State Handler for keeping the application informed of their status. All components also rely on the User component for authentication purposes, as well as context. Components needing file managing, for either uploading or just showing, always rely on the Uploader Widget.

Functionalities from ReactJS such as mix ins, were taken advantage of to reduce boilerplate coding, for API access for example, table handling (such as filtering and ordering) and generic store, actions and components aspects.

State Machine Widget

One of the most important things in a workflow manager is obviously the way we create the workflows. It is crucial on any workflow management system to have an easy way to manage its workflows. One of the most important parts of a workflow design is the state machine editor, since it describes how the states relate to each other.

We are striving for the best possible user experience, so when we realized there were no solid and open-source state machine visual editors written in JavaScript that offered serialization, and even though it is a major effort, we immediately decided to implement one.

Below, on Figure 28, we can see our state machine widget running.

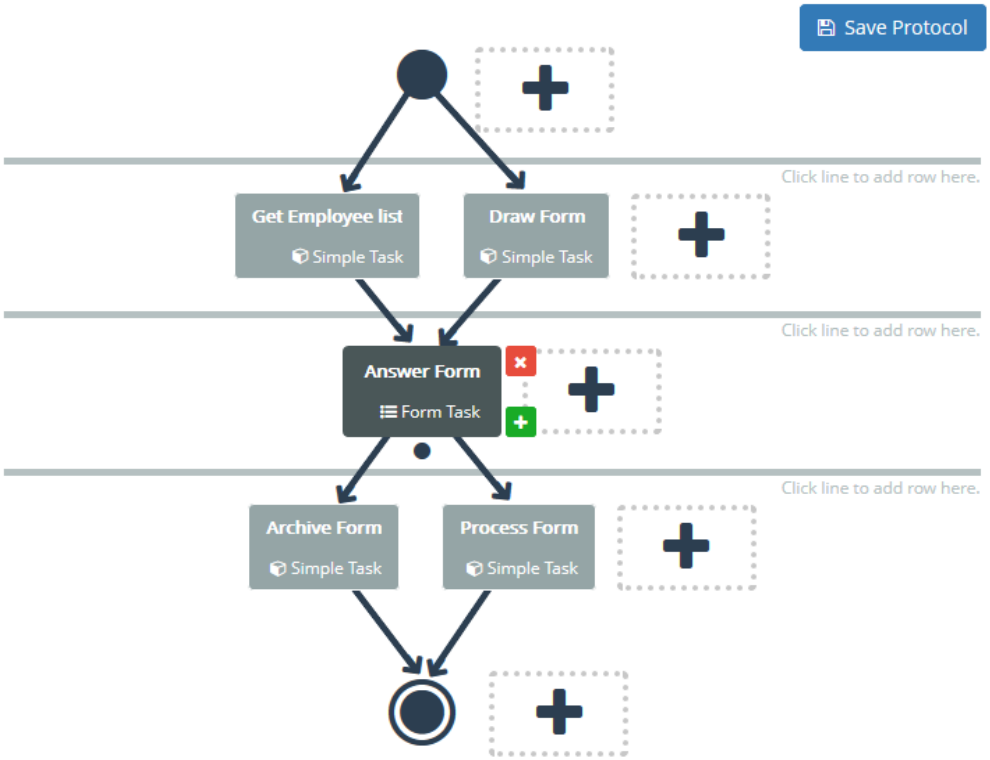

Figure 28 - State machine widget running

The result is an attractive and versatile standalone widget for ReactJS written under the same unidirectional dataflow pattern and which eventually will take a life of its own, apart from the main application. The widget allows creating workflows by directly manipulating a state machine diagram using drag-and-drop besides other direction interactions. The widget features are: multiple class-based state types, that allows users to implement their own types of states by simply extending the default state class; read-only mode; detail panel for state configuration, which can dynamically change between popup, left panel or bottom panel; event handlers such as validate; undo and redo stack.

We use this widget all over the default client that we implemented, ranging from workflow creation, to process displaying.

Uploader Widget

Another very import part of any workflow management system is the file manager. All workflow processes will eventually involve files. Tasks can pass down files or files are part of the task description.

Fortunately, we could indeed find in Filedrop.js a good solution for a file uploading API for JavaScript, which we already boarded in the previous section. However, Filedrop.js does not provide a visual interface for managing the files, it instead provides only a programmatic API, so we decided to implement a widget for managing the file uploads visually using Filedrop.js as the base.

The result is the Uploader Widget, which makes use of Griddle and Filedrop.js, to allow users to manage their files with ease and that even support commentaries. We use this widget in a number of places, like the state detail view and the result answering view.

On Figure 29, we can see the uploader widget running.

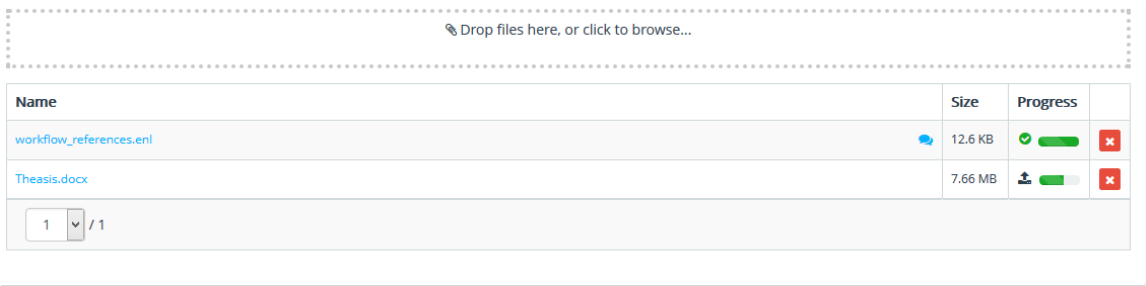

Figure 29 - Uploader widget running

API Connector and web services consumption.

Whenever building a client application that consumes Web services as the source of information, the most vital component is the code that handles the connection to the web services. Without it the application would never work.

As neither ReactJS nor Flux offer the mechanism to communicate with the web services and consume them, given the straightforward REST API from the backend, we built a series of components based upon jQuery Ajax mechanisms and JavaScript Promises that generalize the API communication so the stores can consume them directly. These components take the form of the API connector. We make all communication with the backend through the API connector.

State Handler

Our default client works as a single entry point application. After the application loads, users make all navigation through the application and never change page, in the traditional sense. That is great because we reduce traffic load and make the application very snappy, but it also carries other consequences such as having state, that carries across all the pages.

Our State Handler manages state. If the page is loading, it is saving some content, or even if there are warnings that should be queued to show, this component handles all of that by using this component to make it cross-page.

User

In our workflow system, we have the notion of user and different roles. As such, we must keep this notion in our client application and somehow maintain session. We make this through the usage of a specific component, which handles the user information maintenance and that other components use to contextualize it, like for example, if the user is logged in or not, or if he is an administrator, or owner of a specific workflow, etc.

This component also handles user registration, password recovery, user login and profile editing.

Workflow

This component is responsible for handling the workflow creation, edition, viewing and running (the way by which it turns into a process). This component allows extension, indirectly by extending the State Machine widget class-based states.

It uses the API connector to get the proper workflow data, the user component to determine the user permissions, the State Machine Widget to show the editor that handles the visual creation, edition and showing and finally the uploader widget, to handle upload of documents related with tasks or show the already uploaded files.

Process

The Process component handles the process management procedures. Through it, users running workflows as processes can have a global view of the state of the process, for example, which tasks have been completed, which are running and what users have completed their part of a task, if multiple users have done it. The interface also allows extending deadlines and cancelling existing users or adding new users.

This component uses the API connector to get the proper process data, the user component to determine the user permissions, the State Machine to show the process view, with colour codes for each task current state and the Uploader Widget to show uploaded files related with tasks.

Task / Result

The "Task and Result" is the component responsible for showing and retrieving the detailed view of a specific task when inserted in a running process, and if available their result. This view allows also answering to a task, evaluating dependencies passed down from previous tasks from which this task depended, as well as see related requests of information made for this task. It is possible to extend this views using class-based react components, which extend the default Task react component.

This component uses the API connector to get the proper process task or result, it uses the User component to determine the user permissions and the Uploader Widget to allow uploading of files related with a given result.

Request

The request component is responsible for handling the request detail views for showing or creating requests related with a given task when inserted into a process context. These requests are an important path of communication between process executioner and process task replier.

This component uses the API connector to get the proper request data and the User component to determine the user permissions.

History

The History component is responsible for handling all the logged actions in the system. The component is read-only, since the backend takes care of history keeping. The history filters in context with the user, and allows filtering by process.

This component uses the API connector to get the proper history data and the User component to determine the user permissions.

5.2.3 Form Tasks Plugin

As an example of possible plugin extendibility, we developed a Form Task plugin, for both backend as well as frontend handling, with the mechanism developed for such Task.

Backend Implementation

As was previously mentioned, a backend extension is very easy to develop. We created a new Django application where models that extended the default Task and Result were added, as well as Serializers and Exporters, all extending the default classes already implemented on the default application. Only adding whatever was

new and needed. This extension is essential for the system noticing how to process this new application correctly and automatically.

A new model and view set was also created for handling Forms, which although was not mandatory in any new plugin, allowed us to reuse form templates across several Tasks. For the model for the form schema, the choice was to use Dobtco Formbuilder default schema JSON template, so we would also integrate it in the frontend.

Frontend Implementation

Although trickier than the backend, on the frontend side a new task implementation requires essentially extending the default *SimpleTask* class that should conform to the State Machine Widget API, as well as extending the default simple result class, which should extend the task ReactJS component.

For this plugin, we also created a new page for managing the forms. Since forms can be created and reused across tasks. We implemented the form visual editor using a modified version of the Dobtco Formbuilder.

5.3 Deployment

Although deployment optimization is not usually one of the worries from a development perspective, it is one of the biggest problems from a system manager perspective.

From the end user point of view, the ease of installation of software can change drastically the opinion about the software quality. As such, it is very important that any applications striving for mainstream and big adoption rates adopt a proactive approach on improving the deployment processes of their applications.

We aimed from the beginning to make it very easy to install and update our software. We do however feature a very complicated decoupled environment, which makes it very complex to deploy.

5.3.1 Software Containers

With the advent of cloud computing, there has been a crescent trend of most applications moving towards virtualization, namely working over virtual machines, instead of physical ones. It is a very commode approach, albeit a very costly one since virtual machines are considered very heavy in hardware requirements, as they require each virtual machine to contain a complete copy of an virtualized operative system, which also takes a lot more space [84].

One great solution for easy deployments that has been growing a lot on recent years is the virtual software container concept. The main difference between the typical virtual machine and a software container, is that containers removes a big chunk of common requirements for all virtual machines by creating virtual layers over the operative system of only things the main system does not already have, effectively reducing the hardware requirements. Data containers offer a very lightweight approach, while keeping most of the advantages of a virtual machine.

On Figure 30, we can see a comparison between containers and virtual machines.

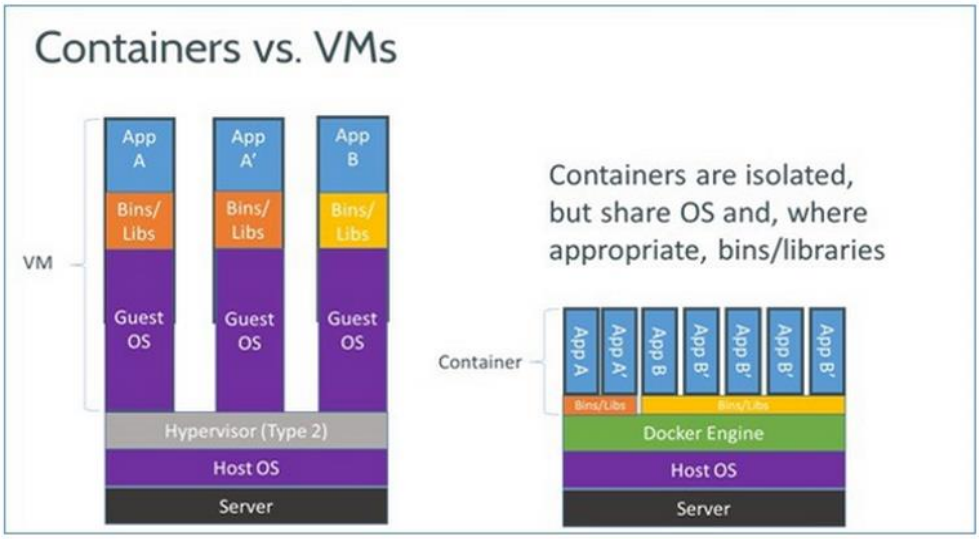

Figure 30 - Comparison between containers and virtual machines [84]

Docker

One of the most well-known software container implementations is Docker [85]. Docker implements the container technology through a repository manager of images and an implementation of containers where layers are stacked upon each other for building an application. We can then build these layered applications upon each other and reuse existing applications to create our own (reducing the necessary work).

Another interesting functionality is the ability of containers to communicate with each other, building a network of containers, which can compose an application.

Docker works almost everywhere and is a great solution for building deploy images of complex software. We use Docker in our application for deploying both our components (backend and default client).

Docker-Compose

As we have seen previously, Docker allows several containers to communicate with each other. It is the way that Docker expects us to use it and complex software's will end up having more than one container. Docker however does not allow us to orchestrate the way containers behave together. It does not make sense to do it manually, so an orchestrator by the name of Docker-Compose, eventually appeared.

Docker-Compose allows running multi-container applications through a single command line, orchestrating the way the containers communicate, and allowing zero configuration deploys. We use Docker-Compose to deploy easily our containers, as we do have multiple containers in the project.

5.3.2 Strategies

Now that we understand what container technology, and mostly what Docker is, we can see and understand the deployment strategy designed for our *Taska* application.

Because of the container technology, *Taska* will run pretty much everywhere, ranging from the default Linux server, OS X, Windows to the more complex Amazon EC2, Azure and other cloud computing platforms. The only requirement is being able to install Docker and support is very wide and always increasing.

Over Docker we require basically two containers, the database PostgreSQL container, and the Taska application container. There two containers are tied together by the Docker-Compose configuration file which tells how these two are related. It is in this configuration file where we can make small-grained configurations. The PostgreSQL container is optional and we can remove it in this file, for large production deployments that require more stability, or external PostgreSQL deploys.

Taska Container

If we open the container apart, to see how it works internally, we can see three main components: the uWSGI, the Nginx, and Celery.

uWSGI and Nginx

Although it has an internal development server, Django itself has no way to serve its content on a production scale. Web page serving is usually done through a web server that pickups the pages and serves the result.

One very well-known web server is Nginx [86]. Nginx is very robust, and allow us to serve web content efficiently. It does however not execute code to generate the pages it serves, as it is not an application server.

Pages must be static, so we need a way to generate these static pages from our Django content. This can be done through uWSGI [87]. uWSGI is an application container for Python code, that generates static content from our Django code, this application container is compatible with Nginx. We use Nginx and uWSGI, as the medium to get our Django project into the web, efficiently.

Celery

As was explained previously in the Architecture chapter, we are using it to handle background asynchronous tasks, such as emailing. We achieved the connection between the uWSGI and Celery through Transmission Control Protocol (TCP).

Overview

On Figure 31, we can see a diagram, which shows all previously explained components, in a much-resumed form. We can also see that we have a documentation pages served through Nginx, which we did not mention before, we generate this documentation using the previously mentioned Sphinx application.

The TCP port we serve our content is configurable on the *docker-compose.yml*, by default we serve it on port 80. Users do need to configure the email server, if they want to use the email functionalities, which we configure on the same file.

The container should start and stop through the proper mechanisms on the Makefile, which adapt the docker-compose syntax into an easier format. We can start the project by just running on the command line **make && make run,** from the main project page**.**

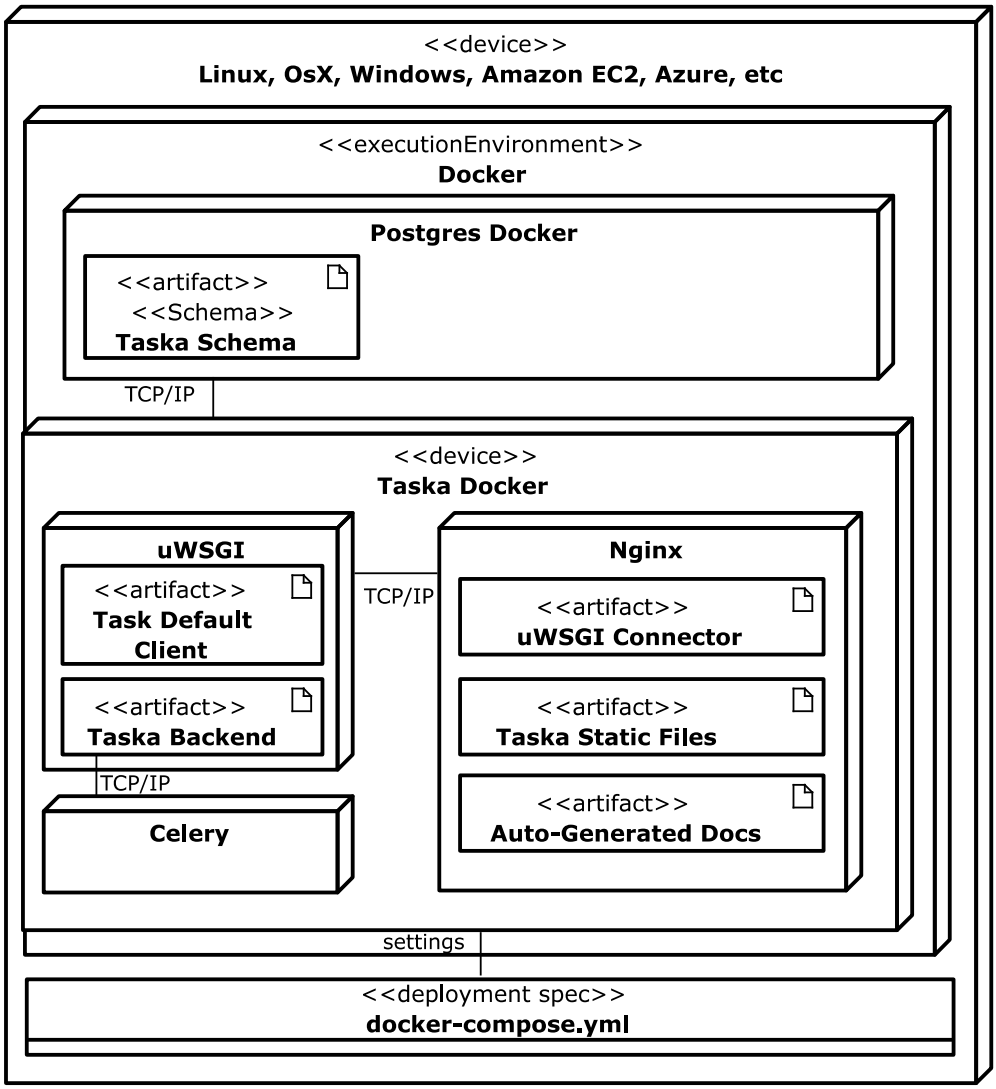

Figure 31 - Typical Taska deploy

5.4 Summary

In this chapter we have a delineation of the proposed architecture for our system as a decoupled solution based on a SaaS architecture with the front-end developed as a web-service consuming platform. We board technologies used in both cases, the architecture defined and describe the web services implemented.

We also board concepts about deployment and deployment strategies designed for fast and easy deploys of our application.

6 Results and Discussion

In previous chapters, we looked at the project from a technical and architectural point of view, discussing how we built the platform, what we used as the building blocks and how all the blocks communicate to work with each other. We even explained how everything could be easily deployed into production environments.

In this chapter, we look at the results of the platform, from an end-user perspective, which is not concerned with architectural considerations, only with usability, appearance and features. To do this, we will analyse the produced deploy for our pilot use-case, used in the EMIF project for the realization of patient register studies. From this perspective, we will analyse the produced user interface and look at way how users' feedback drove the user interface evolution.

6.1 User Interface

The user interface is undoubtedly the most important feature of any application. Users generally evaluate a software's usability and worth by just evaluating the user interface, oblivious about the intrinsic of the implementation. During the default interface development there was a strong focus on maximizing usability and creating an easy to understand, as well as good looking, default interface.

An image is worth than a thousand words, and colours have meaning. Users have an easy time associating colours and icons with predefined actions, so we decided from the get go to use a very flat user interface with the use of strong colours to indicate the several actions possible in each context and heavy use of iconography. In this sense, we represent incremental actions in green, decrement actions or situations where careful attention is due in red and neutral actions in blue. We use yellow to represent relevant and important information/situations or to inform of alterations to current content.

As the user interface was developed mainly with the pilot use-case in mind, which is heavily based upon the run of studies, nomenclature differs a little from the one used on the backend, as such please refer to Workflows being called "Protocols", Processes being called "Studies" and "Process Tasks" being simple called Tasks.

We secure the system through a login user-based mechanism. There is the notion of private user space and to access any of the functionalities the platform provides, the user has first to sign up, and then login. Besides ascertaining identity, user authentication is useful for allowing preferences configuration, which the system allows, such as if the user is interested in receiving history notifications through email or the preferential layout (to optimize based on typical screen resolution access).

We can see the login screen for the frontend interface on Figure 32.

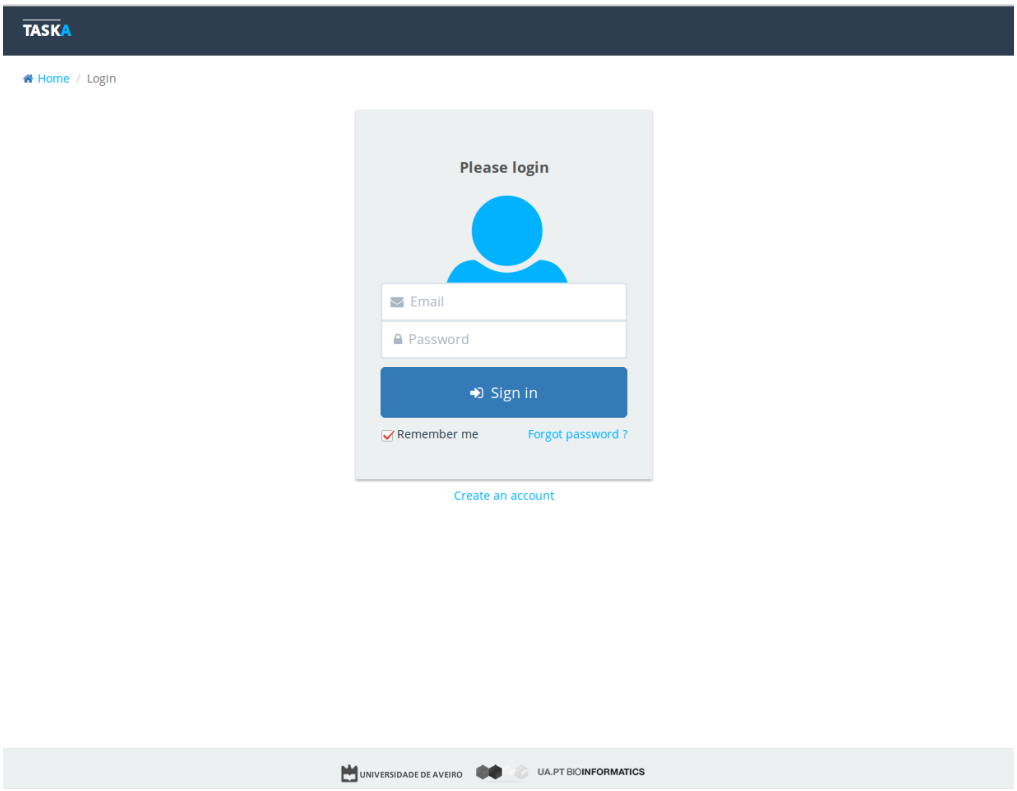

Figure 32 - Login page

From an end-user perspective, the platform is divided in six main areas, which are linked together by a single personal user dashboard which contains all relevant information the system has for the user itself, from a top-down perspective. We can see this dashboard on Figure 33.

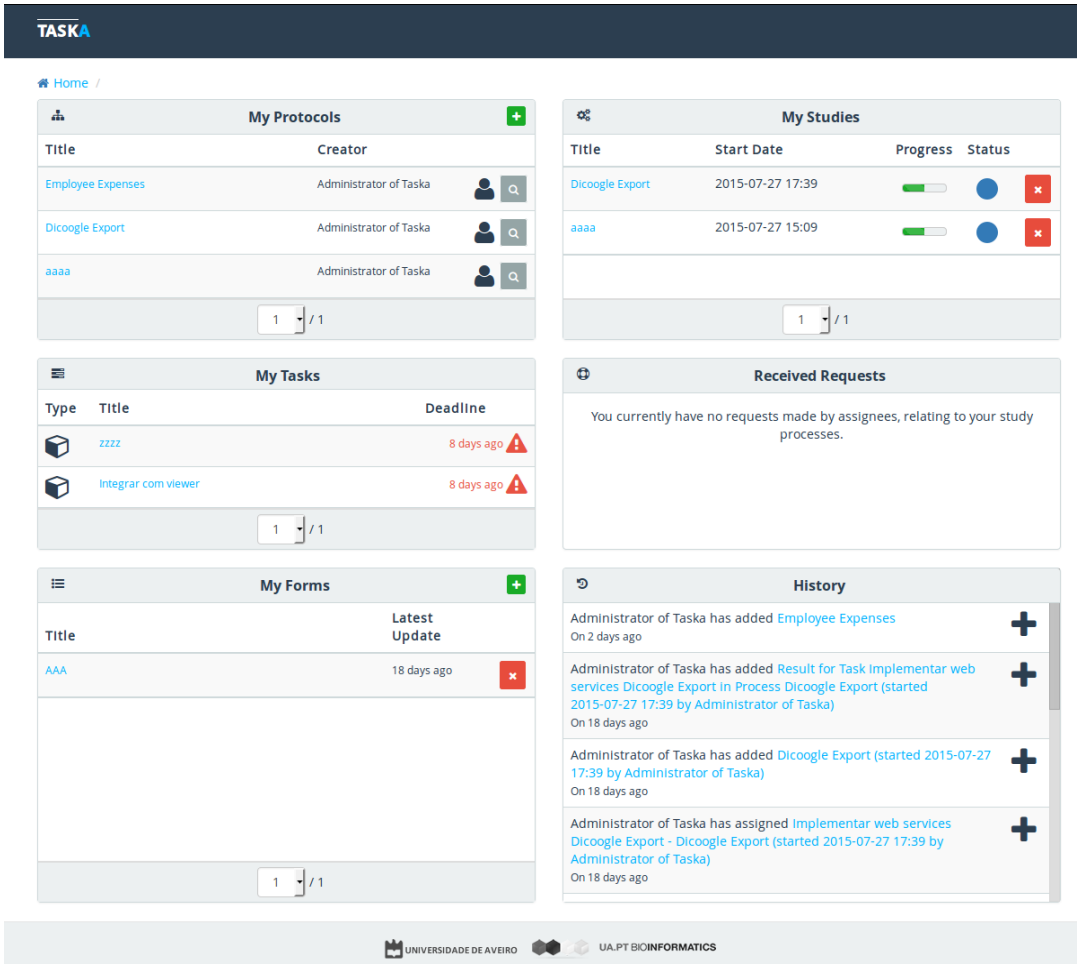

Figure 33 - Personal dashboard

Below, we briefly explain each area and their utility.

6.1.1 My Protocols

The protocol area of the dashboard is the first tier of the system and allows management of the system's basic units, the tasks, and the way they organize themselves into repeatable workflows, which we end up calling a "Protocol". As we seen previously protocols can be public or private, as such the protocol widget on the dashboard shows all public or owned protocols in an easy to navigate list. We mark owned protocols as such by the inclusion of a mark icon accompanied by the name of the creator.

Through this list, we can create new protocols by using the green button with plus sign (on the top-right edge of the widget), or simply use/edit existing ones, by using the magnifying glass icon on each protocol in the list.

We greatly facilitate protocol viewing, creation and edition by the usage of the developed state machine editor already boarded in previous chapter.

When a user opens a Protocol, we show a read-only view of the protocol, which allows users to explore the protocol interactively and execute several actions over it, such as duplicating, running, editing or deleting the protocol.

We can see this read-only view on Figure 34.

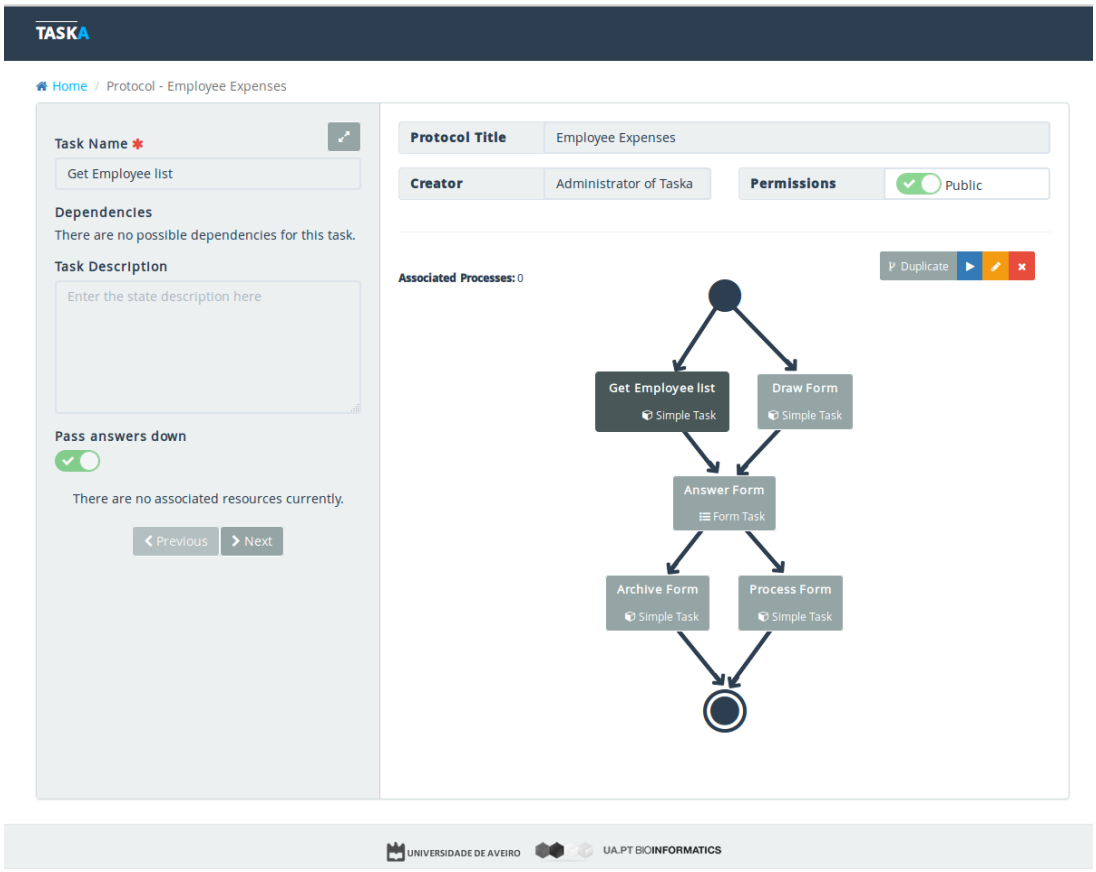

Figure 34 - Protocol read-only view

When editing or creating protocols, users can do it just by drag-and-dropping different types of tasks to the Protocol droppable areas (signed by a plus sign) or clicking the areas themselves and choosing from the type list. The default system supports simple tasks and form tasks, which are associated to a formulary (the form creation belongs to another area, and we detail it in a following sub-chapter).

We structure the protocols in a top-down organization, with the tasks being able to depend upon tasks that are on the tiers above themselves. Users create connections between each task by clicking a task and pulling a line between the tasks.

For each task, several parameters are configurable, such as title, task description, file attachments whether or not dependant tasks will receive the product of the parent task, as well as type of task specific parameter such as the form schema (for form tasks).

On Figure 35, we can see the Protocol editable view.

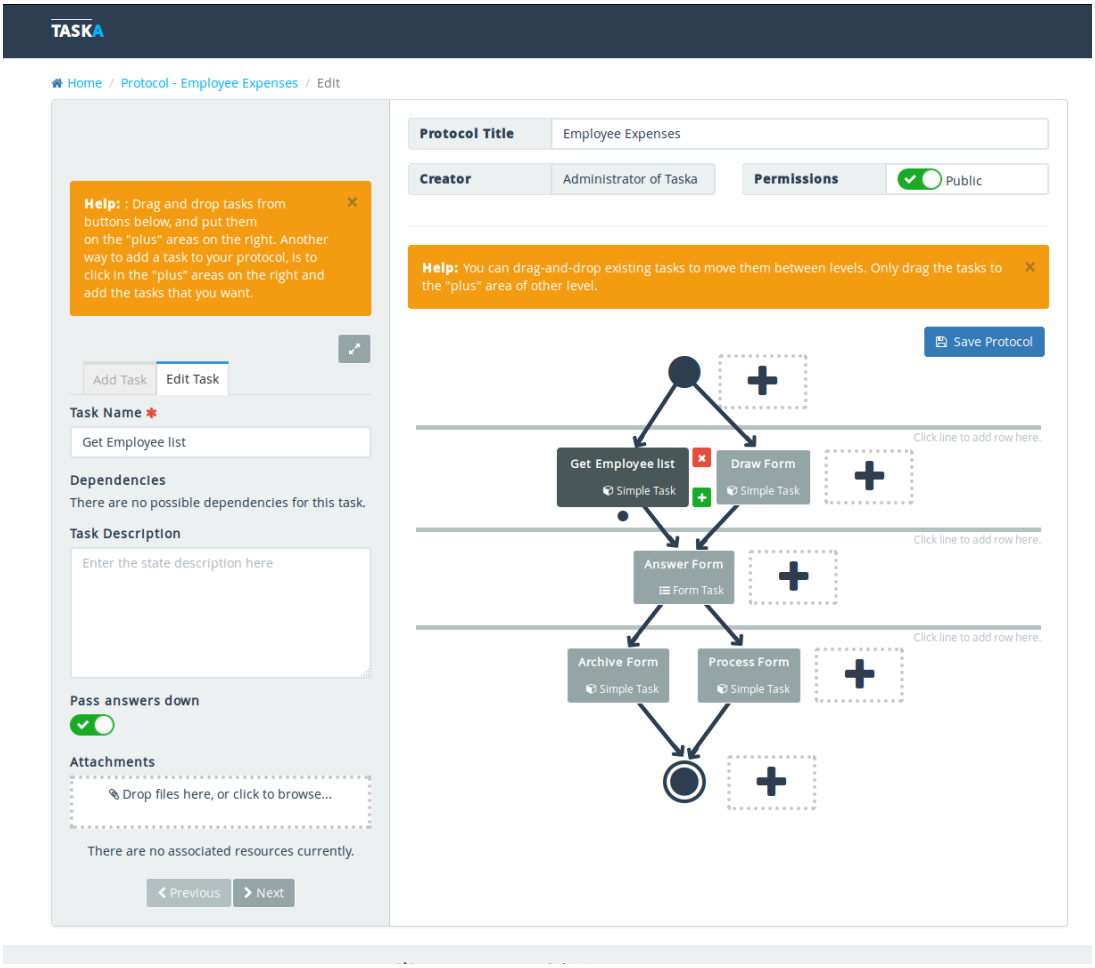

Figure 35 - Protocol edit view

After saving a Protocol, the same protocol becomes runnable. Running a Protocol consists in the act of making a Protocol into a Study. When a Protocol becomes a Study, users specify assignees and deadlines for each task. This separation allows Protocols to run repeatedly, in different occasions by different people.

To run a Protocol a user can just select it from the list of protocols then access the view option by using the magnifying button and clicking the top button.

The Protocol run screen is similar to the read-only screen, allowing viewing the basic information about each task in the protocol. The main difference resides in the ability to allow the specification of several Assignees and a deadline for each task, by clicking each of the tasks.

The process is color-coded. By default all tasks appear grey, an indication of the tasks needing configuration. Whenever we assign a task a deadline and one or more assignees the task will change state and become ready-to-run (signalled by the task becoming white). When all tasks are ready-to-run, a "Run Study" button will become available. When the Protocol runs, the system creates a Study.

On Figure 36, we can see the Protocol running screen.

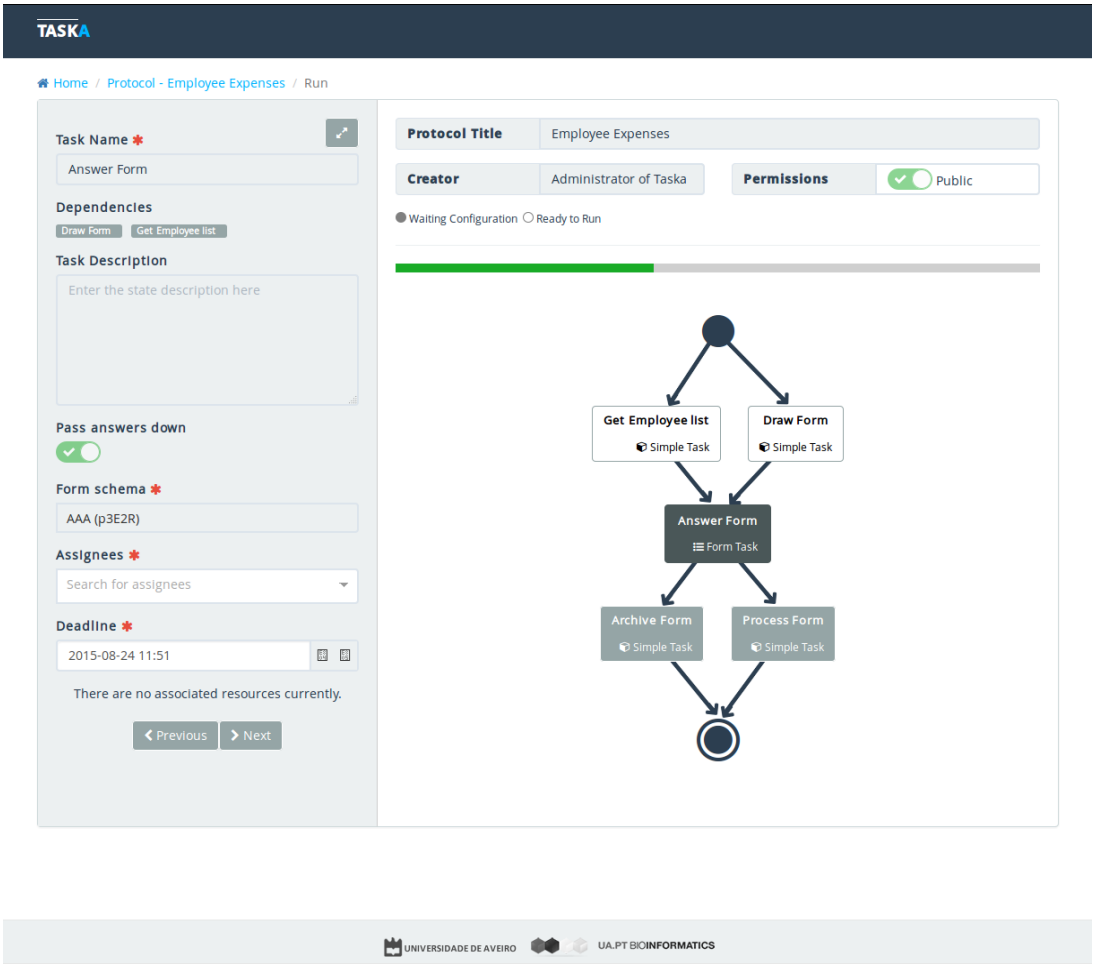

Figure 36 - Protocol running

6.1.2 My Studies

The study area of the dashboard is the second tier of the system and allows users to see studies they are responsible of overseeing. Studies, as seen in previous chapter, are instantiations of Protocols, but with assignees and deadlines.

This list allows easily evaluating the studies progress and status. To view a study details, we can just clock on the study title, which will open the study overview page.

On the study overview page, which is visible on Figure 37, we get a vision similar to the Protocol version, but color-coded with the state each task involved in the study can have. Each task can be waiting for run (if some dependencies are not yet met), running (if they can already be executed), finished, overdue or cancelled.

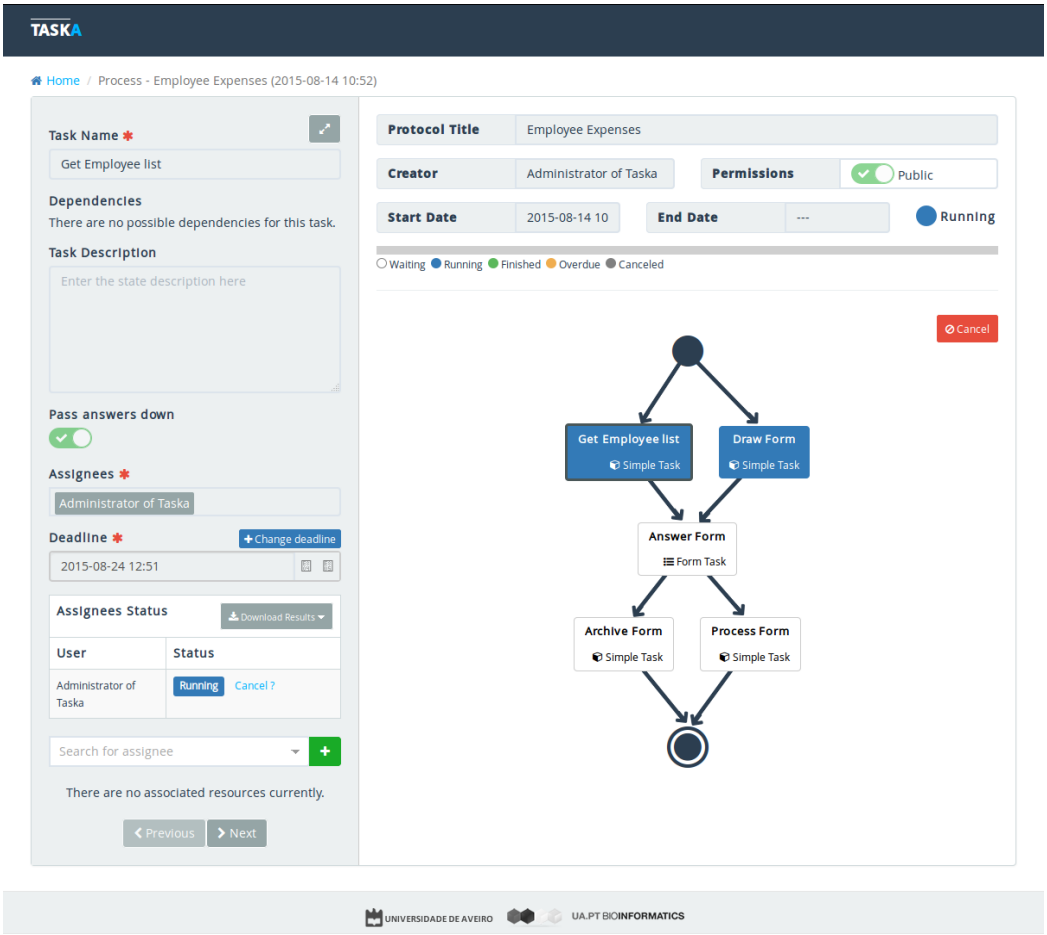

Figure 37 - Study overview

Besides cancelling the study altogether, study overseers can manage each of the tasks individually by clicking on them on the diagram, from there it is possible to remove and add assignees to the tasks as well as changing a task deadline. It is also possible to cancel an individual task, by removing all task assignees.

Tasks carry results to their dependants. We can download individual results from each task details in a variety of formats, such as XLSX and CSV, whenever they are available. When the study has finished running, all results are available by clicking the end state (last double-bordered state). Form tasks will additionally generate a XLSX output from the several form responses.

6.1.3 My Tasks

Tasks are the third tier of the system. Whenever a study is ran by someone, the system attributes tasks to their assignees. From this view, users can manage their assigned tasks. Each task shows the task type (through a task type icon), the title and the deadline. Users can start answering a task by clicking the task title.

Besides completing the answer, which could be just a simple checking it done, or could be answering a form if it is a form task, users can append commentaries and files to the result of the task. We can see on Figure 38 the task replying screen.

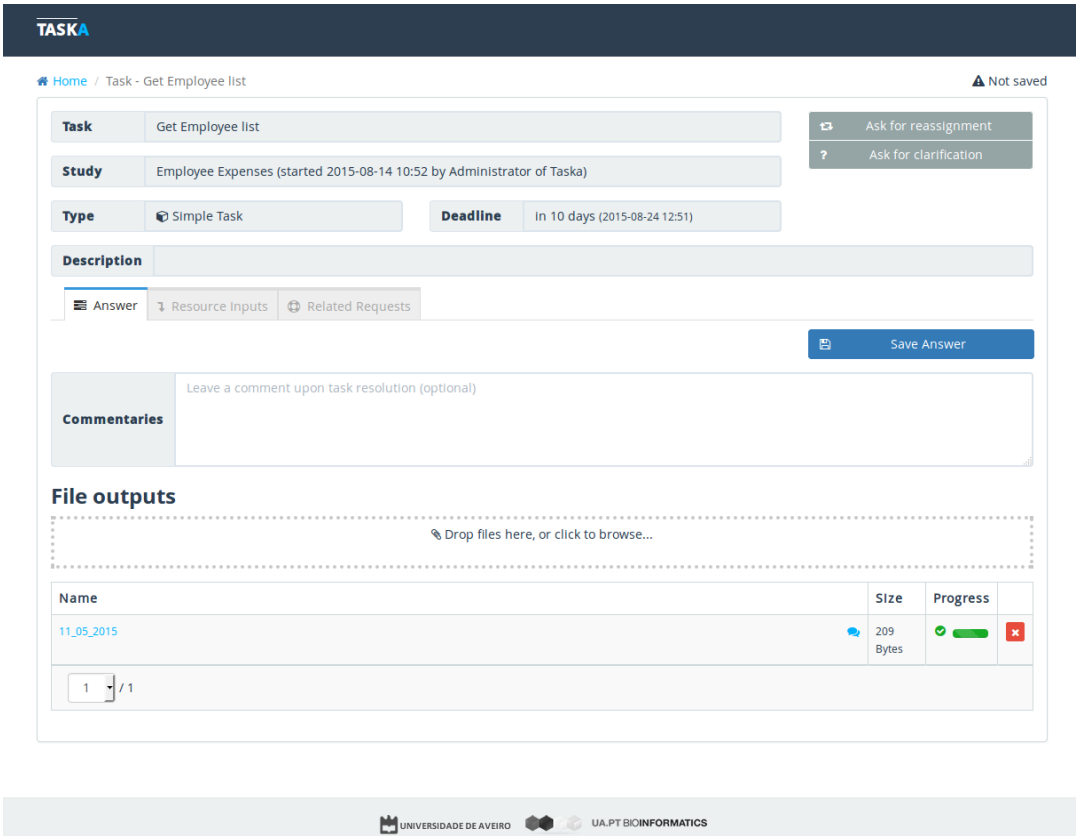

Figure 38 - Task reply

Users can also have access to related files (if the task has associated resources or they are carried as a result from the task dependencies) and check related requests (which could have been done by themselves or other involved users in this particular task) and their answer.

The user can make related requests to the study owner, such as asking for reassignment or clarification of the purposed task.

6.1.4 Requests

The "Received Requests" area shows to the study overseer related requests made by users involved in studies. The request detail page, shown on Figure 39, has information about the request, as well as allowing the response directly through the system. The study overseer has through this area the capability to make the request visible for all task assignees, which could be useful in case other assignees could raise the question again.

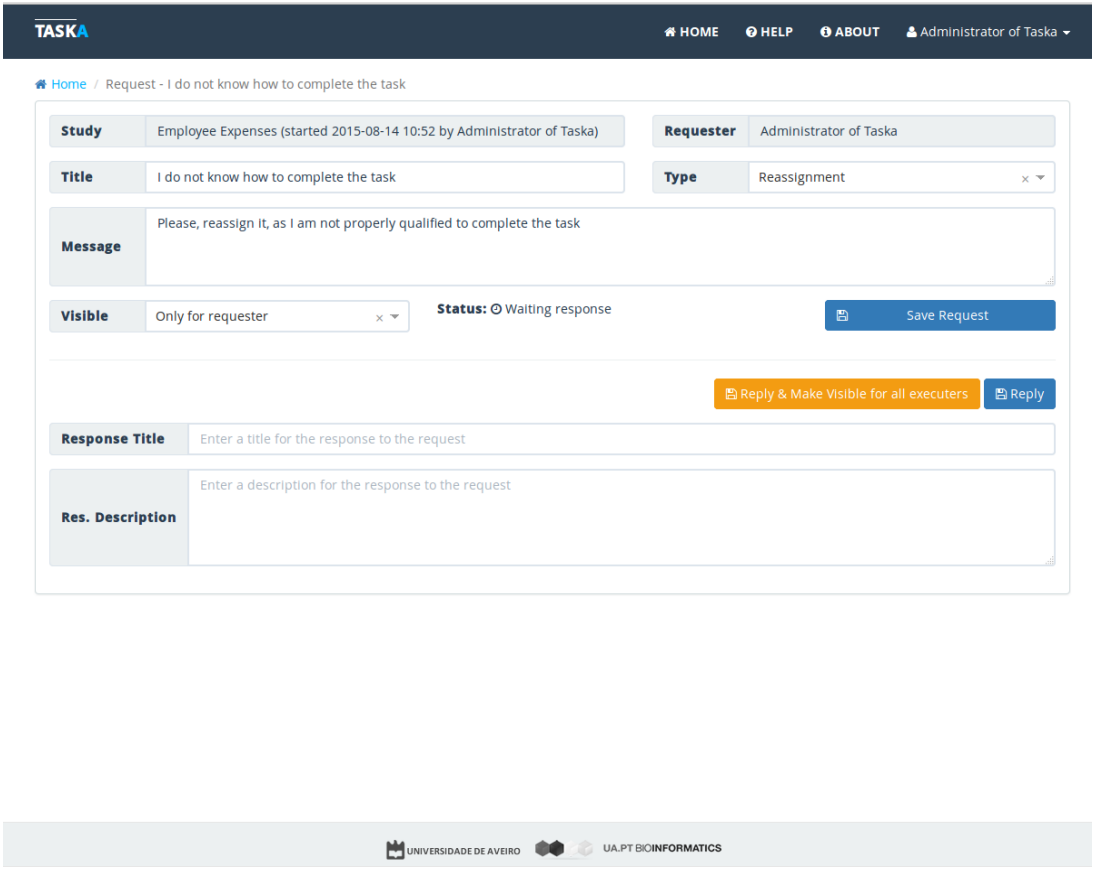

Figure 39 - Requests

6.1.5 My Forms

As we have seen before, one of the type of tasks a Protocol can have is a task with a formulary. Several form tasks can share a unique form schema. This area is where the user is able to see and manage his own form schemas, to later use on Protocol creation. Each form has a title and a latest update date. User can create a formulary schema by clicking the plus green button on the top of the widget and edit by clicking the form name on the form list.

The detailed screen for the form schemas allow the edition/creation through interactive mechanisms, like drag-and-dropping the large array of allowed question types that range from simple text input to multiple choices or dropdowns. Each question is highly configurable, allowing configuration of parameters like for example mandatory filling. We can see this screen on Figure 40.

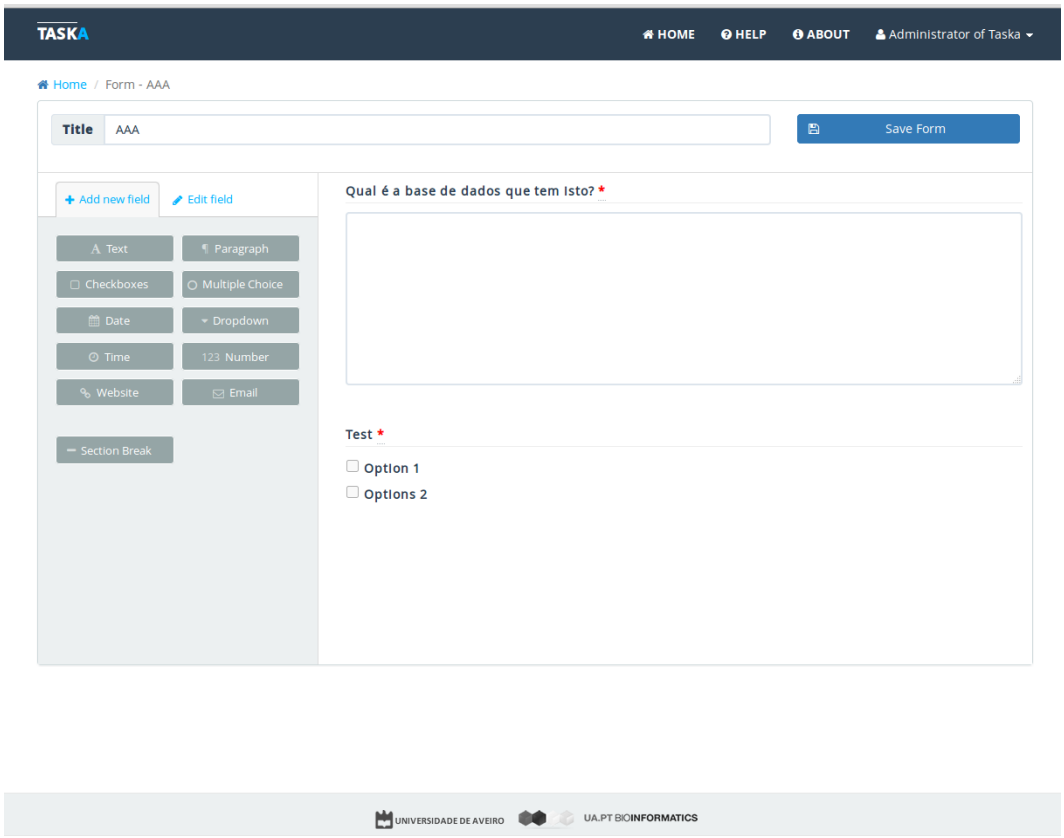

Figure 40 - Form edit

6.1.6 My History

In this area, we can see a feed of all history events that happened in the system, which relate in some way to the logged-in user. The system keeps track of all events on the system, such as protocol creation, study running, task attribution, and task completion. All this is available through an interface shown similar to a timeline, and that users can infinitely browse through.

6.2 User Feedback

During the development cycle there was a strong reliance on user feedback as the engine behind the creative and constructive process for the platform features.

The pilot use-case for Taska, which drove a considerable part of the development process, was the need for a repeatable workflow tool with documentation features inserted in the EMIF project context. Current tools, as seen on the state of the art, did not fulfil properly the needs of the consortium and there was a need for structured documentation but simple to use structure.

There were many interested parties, and a constant flux of feedback and requirements that was incorporated in the tool being developed, with an invite-only beta version even being available for selected consortium members during development. As we expect the tool to continue, evolving based on rolling requirements the consortium presents, long after the scope of this thesis.

In fact, there were some features already developed after the thesis "code freeze", like the possibility of drafting an answer without committing it, start date estimation for tasks, effort indication on tasks, list of schedule tasks or even getting

preliminary inputs for a task without their dependencies having finished, all features originating from consortium feedback.

6.3 Summary

In this chapter we have seen a step-by-step analysis of the produced front-end layout, with description of available functionalities. We also have given a description of the EMIF use-case and feedback received.

7 Conclusions and Future Work

In the previous chapters, we discussed in detail the system inception and conception, from a technical point of view, focusing on architectural concerns, as well as from the perspective of the end-user, concentrating on usability and functionality capabilities.

In this chapter, we will try to plan the future of the platform as a whole, both inside and outside the main pilot use-case, defining a clear long-term strategy for the continuous development and improvement of our solution.

We also try, based on user feedback and our own ideas, to define what features we could develop that will or could make the system more interesting and functional for users interested in the platform.

7.1 New Features

7.1.1 Short term

In the short term or since the code freeze, the following features were/will be developed:

Draft Answers

Allowing users to draft an answer, but not submitting it immediately, keeping it into a draft state, which assignees could complete later. This allows user to build answers gradually, instead of at a unique point in time.

Task Associated Efforts

Associate a determined effort to a task, like for example, a number of hours needed to complete it.

Indication of Assigned Tasks date (even if estimated)

Try to guess when a task will probably be started, based upon dependant deadlines and results delivered.

Allowing access to scheduled tasks

Allowing users to see tasks that are waiting dependencies completion.

Preliminary inputs

Seeing preliminary inputs on scheduled tasks, as assignees gradually fulfil them.

List of accomplished tasks

List of tasks that were already completed, and the given answer, for posterior review by the assignees.

Form duplications

Allowing users to duplicate an existing form, similar to what is possible already on workflow templates.

Quick study repeat

Allow users to quickly repeat a study by defining a time alpha, reusing all the temporal constraints and intervenient in an already complete study.

7.1.2 Long-term

Global Workspace per workflow

The main idea is to allow users participating into a workflow, to have access to a common and referable workspace, where they have resources relevant for all workflow and not just for the assigned tasks.

SSO and/or Social authentication

Social integration is very popular in current times. Most users have already valid identities online, certifiable through well-known single sign-on platforms like Google Accounts or Facebook, the concept is to allow users to authenticate using this methods, instead of requiring them to sign up to our platform exclusively.

Matrix Formulary Task Type

Currently the system already supports formulary tasks, what the system does not supporting is allowing the user to answer several times to the same formulary, in the same task. This could be very relevant for repetitive data collection.

Collaborative Task Type

Sometimes several users should complete a task, but it should be the same answer, not several answers separately. As such, we should develop a new task type, where the answer several assignees construct it simultaneously.

Notifications aggregation

Currently, and as boarded extensively on the architecture of the system, the system register notifications individually for each event that happens on system, being communicated individually both in the log, and if the user has email notifications in the email client. The user however can consider this approach very descriptive. Sometimes it could be desirable for the system to aggregate notifications somehow, and displayed as a single event, or a collection of a single event.

Study Observers

Only users running a workflow have the full picture of a given workflow. Sometimes, overseers could desire to give access to other users to this information, in the role of an observer.

User Groups

Whenever we are referring to a user in the system, we are referring to individual identities. It could however be helpful to refer to a group of users as a unit, to save time in for example, managing assignments.

Text output type

Right now tasks carry outputs, and receive inputs, but even though the system is generic and classify them as resources, they are only of the type file. It could however by useful to carry text IO instead, which users could refer to, or reuse down the line dynamically.

7.2 Long-term strategy

Although we developed the system with a very specific use-case in mind, we designed everything to be generic enough so that the system can adapt to a variety of contexts requiring structured workflow execution. Enlarging of usability scope should be the project main agenda.

The typical deploy of the system will probably be Intranet networks, inside companies or institutions large enough to require workflow maintenance for efficient work execution. We should however not exclude the possibility of having a public unique centralized deploy online, which could be useful for work at a distance or smaller companies where the data being collected is not so sensible.

The biggest problem of a project of this genre is its financial sustainability on the long run. The code is free, and as such, we could not ask users to pay per access. If we take in consideration the main typical predicable deploy, project rent ability could however be reached through the selling of support services for the platform, as well as possibly advertisements in the online hub deploy.

7.3 Final Considerations

In conclusion, we believe the platform developed fulfils the documentation and structural requirements of the EMIF use cases, much better than other state of the art tools, which focus either on individual task management or on structured scientific automated workflow execution.

We believe that our hybrid approach of an object-documented system, with capabilities of a state machine structured tool was a bet that payed off. The decoupled nature of the architecture, which allows both a well-designed and easy to use interface, as well as machine integration if needed, brings the best of both worlds, in a simple and elegant platform.

The several interactive capabilities of the developed reactJS user interface, such as the drag-and-drop state machine editor, bring a touch of user-friendly approach into the equation. Moreover, this interface shows the capability of the platform as a whole, as we developed it based upon the platform decoupled web services, and anyone can developer other interfaces based upon it.

There is still a lot that we can develop around both the platform as well as the interface that could potentially improve upon the existing solutions, increasing further the scope of usability of the system.

7.4 Summary

In this chapter, we have seen planned features for the platform, both on the short as well as the long term. We also looked on possible long-term strategies for the platform and get final considerations about the work completed.

8 References

[1] Eurostart, "Computers and the internet: enterprises - summary of EU aggregates (NACE Rev. 1.1 activity)," ed: Eurostat, 2014.
[2] *Bitrix, Inc*. Available: http://www.bitrixsoft.com
[3] J.-P. Lang. (15-01-2015). *Redmine*. Available: http://www.redmine.org/
[4] K. M. v. H. Wil van der Aalst, *Workflow Management: Models, Methods, and Systems*, 2004.
[5] M. Weske and SpringerLink (Online service). (2007). *Business Process Management Concepts, Languages, Architectures*. Available: http://dx.doi.org/10.1007/978-3-540-73522-9
[6] D. G. M. Hornick, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases,* pp. 119-153, 1995.
[7] (2009). *Business Process Management Glossary*.
[8] *Change Management Control*. Available: http://www.slam-energy-software.com/features-ct
[9] *Taverna - open source and domain independent Workflow Management System*. Available: http://www.taverna.org.uk/
[10] T. e. foundation. (15/01/2015). *Stardust - Comprehensive Business Process Management*. Available: https://www.eclipse.org/stardust/
[11] Oracle. (2003, 2015). Oracle Workflow API Reference. Available: http://docs.oracle.com/cd/B13789_01/workflow.101/b10286/wfapi.htm
[12] *Activiti BPM Platform*. Available: http://activiti.org
[13] *Fireworks Documentation*. Available: http://pythonhosted.org/FireWorks
[14] *FOXopen - The open-source workflow solution*. Available: http://www.foxopen.net
[15] *Go Flow Documentation*. Available: https://code.djangoproject.com/wiki/GoFlow_DocFr
[16] *ViewFlow - reusable django framework*. Available: http://viewflow.io
[17] *Yawl*. Available: http://www.yawlfoundation.org/
[18] R. Hat. (15-01-2015). *jBPM - Open Source Business Process Management*.
[19] A. B. M. Adam A Hunter, Tamas O Szabo, Crispin A Wellington and Matthew I Bellgard. *Yabi*. Available: https://bitbucket.org/ccgmurdoch/yabi/wiki/Home
[20] *Galaxy Project*. Available: https://wiki.galaxyproject.org/
[21] (15-01-2015). *Project Pier*. Available: http://www.projectpier.org/
[22] L. Taiga Agile. (15-01-2015). *Taiga.io*. Available: https://taiga.io/
[23] T. K. M. Saeki, "Software Development Based on Software Pattern Evolution," *Sixth Asia Pacific Software Engineering Conference - APSEC'99,* 1999.
[24] M. Aoyama, "Evolutionary Patterns of Design and Design Patterns " *IEEE*, 2001.
[25] S. H. V. Corrêa, "Software Pattern Communities: Current Practices and Challenges," *CSE Technical reports*, 2007.
[26] M. Richards, *Software Architecture Patterns*: O'Reilly Media, Inc., 2015.
[27] A. A. M. Luo, "Patterns: Service-Oriented Architecture and Web Services," *IBM RedBooks*, 2004.
[28] M. Fowler. (2006). *GUI Architectures*. Available: http://martinfowler.com/eaaDev/uiArchs.html
[29] R. Frey, "MVC Process," MVC-Process.png, Ed., ed, 2010.
[30] B. Kohan. *Guide to Web Application Development*. Available: http://www.comentum.com/guide-to-web-application-development.html
[31] Oracle. *Java Platform, Enterprise Edition: Your First Cup: An Introduction to the Java EE Platform*. Available: https://docs.oracle.com/javaee/7/firstcup/java-ee001.htm
[32] N. Gerard, "Java EE - (J2EE|JEE) Platform," ed, 2009.
[33] Microsoft. *Introduction to ASP.NET Web Forms*. Available: http://www.asp.net/web-forms/what-is-web-forms

[34] S. Hanlseman. *Making Web APIs with ASP.NET MVC*. Available: http://www.hanselman.com/blog/content/binary/Windows-Live-Writer/Making-Web-APIs-with-ASP.NET-MVC-4-Beta_BEA5/image_04e87be2-544a-4de4-bcdd-e97d206c45bb.png
[35] P. Framework. *Frequently Asked Questions*. Available: https://www.playframework.com/documentation/1.1.1/faq
[36] L. C. L. Richardson, *Ruby Cookbook*, 2006.
[37] D. Carta, "Stop Flipping the bird," ed, 2008.
[38] K. Finley, "Twitter Engineer Talks About the Company's Migration from Ruby to Scala and Java," in *readwrite*, ed, 2011.
[39] D. S. Foundation. *Why Django?* Available: https://www.djangoproject.com/start/overview
[40] D. O. Pérez, "¿De qué está hecho chattyhive?," ed, 2014.
[41] R. Branas, *AngularJS Essentials*: Packt Publishing, 2014.
[42] C. R. S. Gribble, "Isolating web programs in modern browser architectures," presented at the Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, 2009.
[43] Google. *Why AngularJS?* Available: https://angularjs.org/
[44] A. Davis, "Implementing a flowchart with SVG and AngularJS," ed, 2014.
[45] I. Tilde. *Ember: Core Concepts*. Available: http://guides.emberjs.com/v1.12.0/concepts/core-concepts
[46] U. Shaked. (2014). *AngularJS vs. Backbone.js vs Ember.js*. Available: https://www.airpair.com/js/javascript-framework-comparison
[47] J. Knebel. (2013) An In-Depth Introduction To Ember.js. Available: http://www.smashingmagazine.com/2013/11/07/an-in-depth-introduction-to-ember-js/
[48] DocumentCloud. *Backbone.JS - Getting Started*. Available: http://backbonejs.org/
[49] I. Facebook. *React- A Javascript Library for building user interfaces*. Available: http://facebook.github.io/react/
[50] I. Facebook. *Flux - Overview*. Available: https://facebook.github.io/flux/docs/overview.html
[51] M. Brassman. (2015). *RefluxJS - Readme*. Available: https://github.com/spoike/refluxjs
[52] RisingStack, "The React.js Way: Getting Started Tutorial," ed, 2015.
[53] T. Christie. *Django REST Framework*. Available: http://www.django-rest-framework.org/
[54] Celery. *Celery: Distributed Task Queue*. Available: http://www.celeryproject.org/
[55] E. Gazoni. *openpyxl - A Python library to read/write Excel 2007 xlsx/xlsm files*. Available: https://openpyxl.readthedocs.org/en/latest
[56] G. Brandl. *Sphinx - Python Documentation Generator*. Available: http://sphinx-doc.org
[57] P. G. D. Group. *PostgreSQL*. Available: http://www.postgresql.org/about
[58] Sentry. *Sentry*. Available: https://www.getsentry.com/welcome
[59] html2text. *html2text*. Available: https://github.com/aaronsw/html2text
[60] *Bootstrap*. Available: http://getbootstrap.com
[61] D. Gandy. *Font Awesome - The iconic font and CSS Toolkit*. Available: http://fortawesome.github.io/Font-Awesome/
[62] j. Foundation. *jQuery - write less, do more*. Available: https://jquery.com
[63] j. Foundation. *jQuery User Interface*. Available: https://jqueryui.com/
[64] I. Department of Better Technology. (2013). *FormBuilder*. Available: https://github.com/dobtco/formbuilder
[65] I. Department of Better Technology, "FormRenderer," 2014.
[66] T. W. I. Chernev. *MomentJS - Parse, validate, manipulate and display dates in JavaScript*. Available: http://momentjs.com
[67] R. F. M. Jackson, "React-Router," 2014.
[68] R. Lanciaux. (2015). *Griddle*. Available: http://griddlegriddle.github.io/Griddle/
[69] S. A. Robbestad. (2015). *React-Breadcrumbs*. Available: https://github.com/svenanders/react-breadcrumbs
[70] G. Mailer. (2014). *React-HotKey*. Available: https://github.com/glenjamin/react-hotkey
[71] J. Watson. (2015). *React-Select*. Available: https://github.com/JedWatson/react-select
[72] *React-SimpleTabs*. Available: https://github.com/pedronauck/react-simpletabs
[73] "React-Toggle," 2015.
[74] J. Quense. (2014). *React-Widgets*. Available: https://github.com/jquense/react-widgets
[75] M. Robenolt. (2014). *Raven-js*. Available: https://github.com/getsentry/raven-js
[76] A. Weber, "FileDrop.js - Self-contained cross-browser JavaScript file upload."
[77] S. McKenzie. (2014). *Babel - Javascript Compiler*. Available: https://babeljs.io/
[78] I. Z. Schlueter. *NPM - Node Package Manager*. Available: https://www.npmjs.com/

[79] N. j. Foundation. *NodeJS - Platform for easily building fast, scalable network applications*. Available: https://nodejs.org
[80] J. Halliday. *Browserify*. Available: http://browserify.org
[81] J. Halliday. *Watchify*. Available: https://github.com/substack/watchify
[82] M. Bazon. *UglifyJS*. Available: https://github.com/mishoo/UglifyJS
[83] F. Marcia. *UglifyCSS*. Available: https://github.com/fmarcia/UglifyCSS
[84] S. J. Vaughan-Nichols. (2014, What is Docker and why is it so darn popular? Available: http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/
[85] I. Docker. *Docker - Build, Ship, Run*. Available: https://www.docker.com
[86] I. Sysoev, "Nginx."
[87] uWSGI. *The uWSGI project*. Available: https://uwsgi-docs.readthedocs.org/en/latest/

9 Appendix

9.1 /api/account

9.1.1 /api/account/ - Method GET

ORDERING FIELDS

```
username, first_name, last_name, email, is_staff, last_login, id
```

FILTERING FIELDS

```
username, first_name, last_name, email, is_staff, last_login, id
```

OUTPUT


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "username": "admin",
            "first_name": "Administrator",
            "last_name": "of Task",
            "email": "email@example.com",
            "is_staff": true,
            "last_login": "2015-08-06 08:48",
            "fullname": "Administrator of Taska",
            "id": 1,
            "profile": {
                "detail_mode_repr": "Detail views appear on the left in the page
sequence",
                "detail_mode": 2,
                "notification": true
            }
        }
    ]
}
```

9.1.2 /api/account/ - Method POST

INPUT

```
{
    "username": "new_user",
    "first_name": "John",
    "last_name": "Doe",
    "email": "me@email.com",
    "is_staff": false,
    "profile": {
        "detail_mode": 1,
        "notification": false
    }
}
```

9.1.3 /api/account/<hash>/ - Method GET

OUTPUT

```
{
    "username": "new_user",
    "first_name": "John",
    "last_name": "Doe",
    "email": "me@email.com",
    "is_staff": false,
    "last_login": "2015-08-06 08:48",
    "profile": {
        "detail_mode": 1,
        "notification": false
    }
}
```

9.1.4 /api/account/<hash>/ - Method PATCH

INPUT

```
{
    "username": "new_user",
    "first_name": "John",
    "last_name": "Doe",
    "email": "me@email.com",
    "is_staff": false,
    "profile": {
        "detail_mode": 1,
        "notification": false
    }
}
```

9.1.5 /api/account/<hash>/ - Method DELETE

INPUT

```
{}
```

9.1.6 /api/account/activate/ - Method DELETE

INPUT

```
{
    "email": "me@email.com"
}
```

9.1.7 /api/account/login/ - Method POST

INPUT

```
{
    "username": "me@email.com",
    "password": "12345"
}
```

9.1.8 /api/account/logout/ - Method GET

OUTPUT

```
{
    "authenticated": false
}
```

9.1.9 /api/account/me/ - Method GET

OUTPUT

```
{
    "username": "new_user",
    "first_name": "John",
    "last_name": "Doe",
    "email": "me@email.com",
    "is_staff": false,
    "last_login": "2015-08-06 08:48",
    "profile": {
        "detail_mode": 1,
        "notification": false
    }
}
```

9.1.10 /api/account/me/ - Method PATCH

INPUT

```
{
    "username": "new_user",
    "first_name": "John",
    "last_name": "Doe",
    "email": "me@email.com",
    "profile": {
        "detail_mode": 1,
        "notification": false
    }
}
```

9.1.11 /api/account/check_email/ - Method POST

INPUT

```
{
    "email": "me@email.com"
}
```

OUTPUT

```
{
    "email": "me@email.com",
    "available": false
}
```

9.1.12 /api/account/register/ - Method POST

INPUT

```
{
    "first_name": "John",
    "last_name": "Doe",
    "email": "me@email.com",
    "password": "12345",
    "profile": {
        "detail_mode": 1,
        "notification": false
    }
}
```

9.1.13 /api/account/recover/ - Method POST

INPUT

```
{
    "email": "me@email.com"
}
```

OUTPUT

```
{
    "success": true
}
```

9.1.14 /api/account/changepassword/ - Method POST

INPUT

```
{
    "email": "me@email.com"
}
```

OUTPUT

```
{
    "success": true
}
```

9.2 /api/resource

9.2.1 /api/resource/ - Method GET

ORDERING FIELDS

```
hash, create_date, latest_update, type
```

FILTERING FIELDS

```
hash, create_date, latest_update, type
```

OUTPUT


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "type": "material.File",
            "creator_repr": "Administrator of Taska",
            "path": "/api/resource/A3jV3/download/",
            "size": 14,
            "hash": "A3jV3",
            "create_date": "2015-07-10T16:05:07.782255Z",
            "latest_update": "2015-07-10T16:05:07.811784Z",
            "filename": "4_jerboa_output2.txt",
            "file": "file/1/A3jV3_VeUxIDZ",
            "linked": false,
            "creator": 1
        }
    ]
}
```


9.2.2 /api/resource/ - Method POST

INPUT


```
{
    "type": "material.File",
    "creator": 1
}
```

OUTPUT


```
{
    "type": "material.File",
    "creator_repr": "Administrator of Taska",
    "path": "/api/resource/A3jV3/download/",
    "size": 14,
    "hash": "A3jV3",
    "create_date": "2015-07-10T16:05:07.782255Z",
    "latest_update": "2015-07-10T16:05:07.811784Z",
    "filename": "4_jerboa_output2.txt",
    "file": "file/1/A3jV3_VeUxIDZ",
    "linked": false,
    "creator": 1
}
```


9.2.3 /api/resource/<hash>/ - Method GET

OUTPUT


```
{
    "type": "material.File",
    "creator_repr": "Administrator of Taska",
    "path": "/api/resource/A3jV3/download/",
    "size": 14,
    "hash": "A3jV3",
    "create_date": "2015-07-10T16:05:07.782255Z",
    "latest_update": "2015-07-10T16:05:07.811784Z",
    "filename": "4_jerboa_output2.txt",
    "file": "file/1/A3jV3_VeUxIDZ",
    "linked": false,
    "creator": 1
}
```


9.2.4 /api/resource/<hash>/ - Method PATCH

INPUT


```
{
    "linked": false,
    "creator": 1
}
```


9.2.5 /api/resource/<hash>/ - Method DELETE

INPUT


```
{}
```

9.2.6 /api/resource/<hash>/download/ - Method GET

OUTPUT

Binary data, if of type file, no content if generic Resource. Each type of resource should implement this as they see fit.

9.2.7 /api/resource/<hash>/comment/ - Method GET

OUTPUT


```
{
   "comments": [
      {
         "user_repr": "Administrator of Taska",
         "create_date": "2015-08-06T10:52:07.693465Z",
         "latest_update": "2015-08-06T10:52:07.693490Z",
         "comment": "example of a comment",
         "resource": 14,
         "user": 1
      }
   ]
}
```


9.2.8 /api/resource/<hash>/comment/ - Method POST

INPUT


```
{
   "comment": "example of a comment"
}
```

OUTPUT


```
{
   "comment": {
      "user_repr": "Administrator of Taska",
      "create_date": "2015-08-06T10:52:07.693465Z",
      "latest_update": "2015-08-06T10:52:07.693490Z",
      "comment": "example of a comment",
      "resource": 14,
      "user": 1
   }
}
```


9.2.9 /api/resource/my/upload/ - Method POST

INPUT

Binary Data as 'application/octet-stream' with the file name passed through the header HTTP_X_FILE_NAME

9.3 /api/task

9.3.1 /api/task/ - Method GET

ORDERING FIELDS

workflow, hash, sortid, title, description, type

FILTERING FIELDS

workflow, hash, sortid, title, description, type

OUTPUT


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "type": "tasks.SimpleTask",
            "dependencies": [
                {
                    "dependency": "OnGon"
                },
                {
                    "dependency": "lnqZJ"
                }
            ],
            "resources": [],
            "hash": "2nr6n",
            "sortid": 2,
            "title": "Unnamed",
            "description": "",
            "output_resources": true
        }
    ]
}
```

9.3.2 /api/task/ - Method POST

INPUT


```
{
    "type": "tasks.SimpleTask",
    "dependencies": [
        {
            "dependency": "OnGon"
        },
        {
            "dependency": "lnqZJ"
        }
    ],
    "resources": [],
    "sortid": 2,
    "title": "Unnamed",
    "description": "",
    "output_resources": true
}
```


9.3.3 /api/task/<hash>/ - Method GET

OUTPUT


```
{
    "type": "tasks.SimpleTask",
    "dependencies": [
        {
            "dependency": "OnGon"
        },
        {
            "dependency": "lnqZJ"
        }
    ],
    "resources": [],
    "hash": "2nr6n",
    "sortid": 2,
    "title": "Unnamed",
    "description": "",
    "output_resources": true
}
```

9.3.4 /api/task/<hash>/ - Method PATCH

INPUT


```
{
    "type": "tasks.SimpleTask",
    "dependencies": [
        {
            "dependency": "OnGon"
        },
        {
            "dependency": "lnqZJ"
        }
    ],
    "resources": [],
    "sortid": 2,
    "title": "Unnamed",
    "description": "",
    "output_resources": true
}
```


9.3.5 /api/task/<hash>/ - Method DELETE

INPUT


```
{}
```

9.4 /api/workflow

9.4.1 /api/workflow/ - Method GET

ORDERING FIELDS

owner, hash, title, create_date, latest_update, public, searchable, forkable

FILTERING FIELDS

owner, hash, title, create_date, latest_update, public, searchable, forkable

OUTPUT


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "permissions": {
                "public": true,
                "searchable": true,
                "forkable": true
            },
            "tasks": [
                {
                    "type": "tasks.SimpleTask",
                    "dependencies": [],
                    "resources": [],
                    "hash": "lnqZJ",
                    "sortid": 1,
                    "title": "Unnamed 1",
                    "description": "",
                    "output_resources": true
                }
            …
            ],
            "assoc_processes": [],
            "owner_repr": "Administrator of Taska 1",
            "hash": "vnwNR",
            "title": "123321123",
            "create_date": "2015-08-06T09:44:36.424491Z",
            "latest_update": "2015-08-06T09:44:36.540310Z",
            "owner": 1
        }
    ]
}
```

9.4.2 /api/workflow/ - Method POST

INPUT

```
{
    "permissions": {
        "public": true,
        "forkable": true,
        "searchable": true
    },
    "title": "Workflow title",
    "tasks": [
        {
            "sid": 1,
            "title": "Unnamed",
            "type": "tasks.SimpleTask",
            "sortid": 1,
            "description": "",
            "dependencies": [],
            "resourceswrite": [],
            "output_resources": true
        },
        {
            "sid": 2,
            "title": "Unnamed 2",
            "type": "tasks.SimpleTask",
            "sortid": 2,
            "description": "",
            "dependencies": [
                {
                    "dependency": 1
                }
            ],
            "resourceswrite": [],
            "output_resources": true
        }
    ]
}
```

9.4.3 /api/workflow/<hash>/ - Method GET

OUTPUT


```
{
    "permissions": {
        "public": true,
        "searchable": true,
        "forkable": true
    },
    "tasks": [
        {
            "type": "tasks.SimpleTask",
            "dependencies": [
                {
                    "dependency": "OnGon"
                }
            ],
            "resources": [],
            "hash": "2nr6n",
            "sortid": 2,
            "title": "Unnamed",
            "description": "",
            "output_resources": true
        },
        {
            "type": "tasks.SimpleTask",
            "dependencies": [],
            "resources": [],
            "hash": "OnGon",
            "sortid": 1,
            "title": "Unnamed 4",
            "description": "",
            "output_resources": true
        }
    ],
    "assoc_processes": [],
    "owner_repr": "Administrator of Taska 1",
    "hash": "vnwNR",
    "title": "123321123",
    "create_date": "2015-08-06T09:44:36.424491Z",
    "latest_update": "2015-08-06T09:44:36.540310Z",
    "owner": 1
}
```


9.4.4 /api/workflow/<hash>/ - Method PATCH

INPUT

Same as 9.4.2

9.4.5 /api/workflow/<hash>/ - Method DELETE

INPUT

```
{}
```

9.4.6 /api/workflow/<hash>/fork/ - Method GET

INPUT

Same as 9.4.3

9.5 /api/process

9.5.1 /api/process/ - Method GET

ORDERING FIELDS

workflow, hash, start_date, end_date, status, executioner, object_repr

FILTERING FIELDS

workflow, hash, start_date, end_date, status, executioner, object_repr

OUTPUT


```
{
   "count": 1, "next": null, "previous": null,
   "results": [
      {
         "tasks": [
            {
               "task": "NnAO3", "task_repr": "Unnamed",
               "users": [
                  {
                     "user_repr": "Administrator of Taska 1",
                     "result": {
                        "type": "form.FormResult",
                        "process": "7J7GJ", "task": "NnAO3",
                        "user": 1,  "user_repr": "Administrator of Taska 1",
                        "outputs": [],
                        "processtaskuser": "jRlo3",
                        "process_owner": 1,
                        "date": "2015-07-07T16:13:49.966798Z",
                        "comment": "", "hash": "vnwNR"
                     },
                     "reassigned": false, "reassign_date": null,
                     "finished": true,
                     "hash": "jRlo3",
                     "user": 1
                  }
               ],
               "deadline_repr": "2015-07-17T16:25", "status": 3,
               "deadline": "2015-07-17T16:25:00Z",
               "hash": "pRy9R"
            }
         ],
         "workflow": "xRXen",
         "start_date": "2015-07-07 15:25",
         "object_repr": "form test", "progress": 100,
         "status_repr": "Process has ended successfully",
         "hash": "7J7GJ",
         "end_date": "2015-07-07T16:13:49.957631Z",
         "status": 2,  "executioner": 1
      }
   ]
```

9.5.2 /api/process/ - Method POST

INPUT


```
{
    "workflow":"MnN7J",
    "tasks":[
        {
            "users":[
                {
                    "user":1
                }
            ],
            "deadline":"2015-08-16T14:04",
            "name":"Unnamed 2",
            "task":"X3dMR"
        },
        {
            "users":[
                {
                    "user":1
                }
            ],
            "deadline":"2015-08-16T14:03",
            "name":"Unnamed 1",
            "task":"mnK2R"
        },
        {
            "users":[
                {
                    "user":1
                }
            ],
            "deadline":"2015-08-16T14:04",
            "name":"Unnamed",
            "task":"d36mJ"
        }
    ]
}
```

9.5.3 /api/process/<hash>/ - Method GET

OUTPUT


```
{
    "tasks": [
        {
            "id": 18,
            "task": "NnAO3",
            "task_repr": "Unnamed",
            "users": [
                {
                    "user_repr": "Administrator of Taska 1",
                    "result": {
                        "type": "form.FormResult",
                        "process": "7J7GJ",
                        "task": "NnAO3",
                        "user": 1,
                        "user_repr": "Administrator of Taska 1",
                        "outputs": [],
                        "processtaskuser": "jRlo3",
                        "process_owner": 1,
                        "answer": {
                            "c2": "123"
                        },
                        "date": "2015-07-07T16:13:49.966798Z",
                        "comment": "",
                        "hash": "vnwNR"
                    },
                    "reassigned": false,
                    "reassign_date": null,
                    "finished": true,
                    "hash": "jRlo3",
                    "user": 1
                }
            ],
            "deadline_repr": "2015-07-17T16:25",
            "status": 3,
            "deadline": "2015-07-17T16:25:00Z",
            "hash": "pRy9R"
        }
    ],
    "workflow": "xRXen",
    "start_date": "2015-07-07 15:25",
    "object_repr": "form test",
    "progress": 100,
    "status_repr": "Process has ended successfully",
    "hash": "7J7GJ",
    "end_date": "2015-07-07T16:13:49.957631Z",
    "status": 2,
    "executioner": 1
}
```

9.5.4 /api/process/<hash>/ - Method PATCH

OUTPUT


```
{
    "tasks": [
        {
            "id": 18,
            "task": "NnAO3",
            "users": [
                {
                    "user_repr": "Administrator of Taska 1",
                    "result": {
                        "type": "form.FormResult",
                        "process": "7J7GJ",
                        "task": "NnAO3",
                        "user": 1,
                        "user_repr": "Administrator of Taska 1",
                        "outputs": [],
                        "processtaskuser": "jRlo3",
                        "process_owner": 1,
                        "answer": {
                            "c2": "123"
                        },
                        "date": "2015-07-07T16:13:49.966798Z",
                        "comment": "",
                        "hash": "vnwNR"
                    },
                    "reassigned": false,
                    "reassign_date": null,
                    "finished": true,
                    "hash": "jRlo3",
                    "user": 1
                }
            ],
            "status": 3,
            "deadline": "2015-07-17T16:25:00Z",
            "hash": "pRy9R"
        }
    ],
    "workflow": "xRXen",
    "start_date": "2015-07-07 15:25",
    "progress": 100,
    "hash": "7J7GJ",
    "end_date": "2015-07-07T16:13:49.957631Z",
    "status": 2,
    "executioner": 1
}
```

9.5.5 /api/process/<hash>/ - Method DELETE

OUTPUT

```
{}
```

9.5.6 /api/process/<hash>/cancel/ - Method GET

OUTPUT

Same as 9.5.3

9.5.7 /api/process/<hash>/adduser/ - Method POST

INPUT

```
{
   "ptask":"B3k8R",
   "user":"11"
}
```

9.5.8 /api/process/<hash>/canceluser/ - Method POST

INPUT

```
{
   "ptask":"B3k8R",
   "user":"11",
   "val": true // Indicates whether this cancel is result of a reassignment or simple cancelation
}
```

9.5.9 /api/process/<hash>/changedeadline/ - Method POST

INPUT

```
{
   ptask: "EngE3",
   deadline: "2015-07-20T03:51"
}
```

9.5.10 /api/process/my/tasks/ - Method GET

OUTPUT


```
{
    "count": 3,
    "next": null,
    "previous": null,
    "results": [
        {
            "user_repr": "Administrator of Taska 1",
            "task_repr": "Unnamed",
            "type": "tasks.SimpleTask",
            "deadline": "2015-07-17T01:50:00Z",
            "reassigned": false,
            "reassign_date": null,
            "finished": false,
            "hash": "pRy9R",
            "user": 1,
            "processtask": {
                "id": 17,
                "type": "tasks.SimpleTask",
                "process_repr": "321 (started 2015-07-07 00:50 by Administrator of
Taska 1)",
                "parent": {
                    "type": "tasks.SimpleTask",
                    "dependencies": [],
                    "resources": [],
                    "hash": "anVwJ",
                    "sortid": 1,
                    "title": "Unnamed",
                    "description": "",
                    "output_resources": true
                },
                "task": "anVwJ",
                "deadline_repr": "2015-07-17T01:50",
                "status": 2,
                "deadline": "2015-07-17T01:50:00Z",
                "hash": "BJpe3",
                "process": "WJ80J"
            }
        }
    ]
}
```

9.5.11 /api/process/my/task/<hash>/- Method GET

OUTPUT


```
{
    "user_repr": "Administrator of Taska 1",
    "requests": [],
    "task_repr": "Unnamed",
    "type": "tasks.SimpleTask",
    "deadline": "2015-07-17T01:50:00Z",
    "dependencies": [],
    "reassigned": false, "reassign_date": null,
    "finished": false,
    "hash": "pRy9R",
    "user": 1,
    "processtask": {
        "id": 17,
        "type": "tasks.SimpleTask",
        "process_repr": "321 (started 2015-07-07 00:50 by Administrator of Taska 1)",
        "parent": {
            "type": "tasks.SimpleTask",
            "dependencies": [],
            "resources": [],
            "hash": "anVwJ",
            "sortid": 1,
            "title": "Unnamed",
            "description": "",
            "output_resources": true
        },
        "task": "anVwJ",
        "deadline_repr": "2015-07-17T01:50",
        "status": 2,
        "deadline": "2015-07-17T01:50:00Z",
        "hash": "BJpe3",
        "process": "WJ80J"
    }
}
```


/api/process/my/task/<hash>/dependencies/- Method GET

OUTPUT

Same as 9.5.12

/processtask/<hash>/export/<mode>/ - Method GET

OUTPUT

Depends upon the format

9.6 /api/result

9.6.1 /api/result/ - Method GET

ORDERING FIELDS

date, comment, hash, process, task

FILTERING FIELDS

date, comment, hash, process, task

OUTPUT


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "type": "form.FormResult",
            "process": "MnN7J",
            "task": "Vn01J",
            "user": 1,
            "user_repr": "Administrator of Taska 1",
            "outputs": [
                {
                    "type": "material.File",
                    "creator_repr": "Administrator of Taska 1",
                    "path": "/api/resource/A3jV3/download/",
                    "size": 14,
                    "hash": "A3jV3",
                    "create_date": "2015-07-10T16:05:07.782255Z",
                    "latest_update": "2015-07-10T16:05:07.811784Z",
                    "filename": "4_jerboa_output2.txt",
                    "file": "file/1/A3jV3_VeUxIDZ",
                    "linked": false,
                    "creator": 1
                }
            ],
            "processtaskuser": "VJQN3",
            "process_owner": 1,
            "date": "2015-07-10T16:05:13.013547Z",
            "comment": "321",
            "hash": "jJ9kR"
        },
    ]
}
```

9.6.2 /api/result/- Method POST

INPUT


```
{
    "task":"anVwJ",
    "process":"WJ80J",
    "type":"result.SimpleResult",
    "comment":"test",
    "outputswrite": ["vRxg3"]
}
```


9.6.3 /api/result/hash/- Method GET

OUTPUT


```
{
    "type": "form.FormResult",
    "process": "xRXen",
    "task": "NnAO3",
    "user": 1,
    "user_repr": "Administrator of Taska 1",
    "outputs": [],
    "processtaskuser": "jJ9kR",
    "process_owner": 1,
    "date": "2015-07-07T17:38:58.776983Z",
    "comment": "",
    "hash": "MnN7J"
}
```


9.6.4 /api/result/hash/- Method PATCH

INPUT


```
{
    "type": "form.FormResult",
    "process": "xRXen",
    "task": "NnAO3",
    "user": 1,
    "outputs": [],
    "processtaskuser": "jJ9kR",
    "process_owner": 1,
    "date": "2015-07-07T17:38:58.776983Z",
    "comment": "",
    "hash": "MnN7J"
}
```


9.6.5 /api/result/hash/- Method DELETE

INPUT


```
{}
```

9.7 /api/history

9.7.1 /api/history/ - Method GET

ORDERING FIELDS

None

None

FILTERING FIELDS

OUTPUT

9.7.2 /api/history/<model>/<pk>/ - Method GET

Same as 9.7.1